MASTER IN COMPUTER SCIENCE

Concept of virtual machine

Leroy, Patrick

Award date:
1984

Awarding institution:
University of Namur

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX
INSTITUT D'INFORMATIQUE

CONCEPT OF VIRTUAL MACHINE

Mémoire présenté par

Patrick Leroy

en vue de l'obtention
du titre de
Licencié et Maître en informatique

Année académique 1983 - 1984

I would like to thank all those who have supported me when I most need encouragement or help, from my promoter and high school teacher, Mr. Ramaekers; all the SWN22 team members and especially the team leader Mr. de Cocquéau; to my parents without whom this work would not have been possible.

FOREWORD

1 Foreword

This investigation presents a brief State-of-the-Art regarding Virtual Machines concept and their applications.

Section 2 gives a short history of the third generation architecture and the concept of dual state architecture. It also introduces the concept of virtual machine.

Section 3 deals with the analysis of some important aspects of virtual machine such as architecture, main features, types, major features of virtual machines, formal conditions of virtualization.

Section 4 analyses the implications of virtual machines for the computing system. These are located in the fields of integrity, performance and sharing of services.

Section 5 presents the two major applications of virtual machine system.

In section 6, the aspect of performance degradation is deeply studied by means of a practical model. That will result in a statistic study, based on the difference of time requested for the same instruction realized in a real or virtual machine environment.

INTRODUCTION

2 Introduction

2.1 The third generation architecture

In the beginning of the 60's, two major innovations were introduced to improve the performance of a computing system. These were I/O processors and multiprogramming.

As a consequence of the first improvement, computing systems became multiprocessor configurations where non identical processors could have access to the main memory of the system. The second improvement led to several processes sharing a single central processor while vying for a common pool of resources.

These two developments caused some problems with regard to the integrity of the system. For example, an I/O processor executing a wrong channel program could damage areas belonging to other processes or a process executing an "incorrect" procedure of the system could cause the same troubles. Since abundant experiences had demonstrated that it is impossible to rely on the correctness of all software, the multiprogramming/multiprocessing architecture had to found upon a new approach. This one was called "dual state" architecture.

2.2 The dual state architecture

The software was divided into two distinct classes: the first one containing a small amount of code presumed to be correct called the privileged software nucleus and the second one containing all the rest. At the same time, the architecture of the system was defined in such a way that any part of the software which could interfere with other processes belongs to the second class of software.

Essentially, the third generation architecture was found upon two different modes of working (privileged /non privileged , master/slave, system/user, ...etc) to allow the execution of some critical instructions only in the privileged mode. These critical instructions are for example: I/O, memory and interrupt management.

Experience has shown that this solution is very powerful on condition that the privileged nucleus is limited in quantity, stable in the sense that few changes are made over long periods of time, and written by skilled professional programmers.

This new type of architecture has proved its value by fostering the development of computing systems with true simultaneity of I/O operations and high overall resources utilization.

But it has also created new types of problems due to the fact that only the nucleus can access and control all the functions of the hardware.

A first problem is the portability of a program from an environment to another . In fact, from a computer to another, the nucleus can change a lot, even if the hardware of both machines is quite similar. A user wanting to execute a program from one computing system on another has two solutions to solve his problem: either he converts his program or he replaces the first machine nucleus by the second's one. Unfortunately, none of these two solutions is very attractive or swift.

A second problem is the impossibility to run two distinct nuclei at the same time. This makes developments and modifications very difficult because system programmers have to work on a dedicated machine at their disposal, and additionally, all the advantages of the third generation architecture are lost.

A final problem is that test and diagnostic software must have access to and control of all functional capabilities of the hardware and thus cannot be run simultaneously with the privileged nucleus. This severely curtails the amount of testing and diagnoses that can be performed without interfering with normal production schedules.

In conclusion, all the major problems caused by this new type of architecture are due to the fact that it only existed one interface between the hardware and the user's programs: the privileged software nucleus.

2.3 Concept of virtual machine

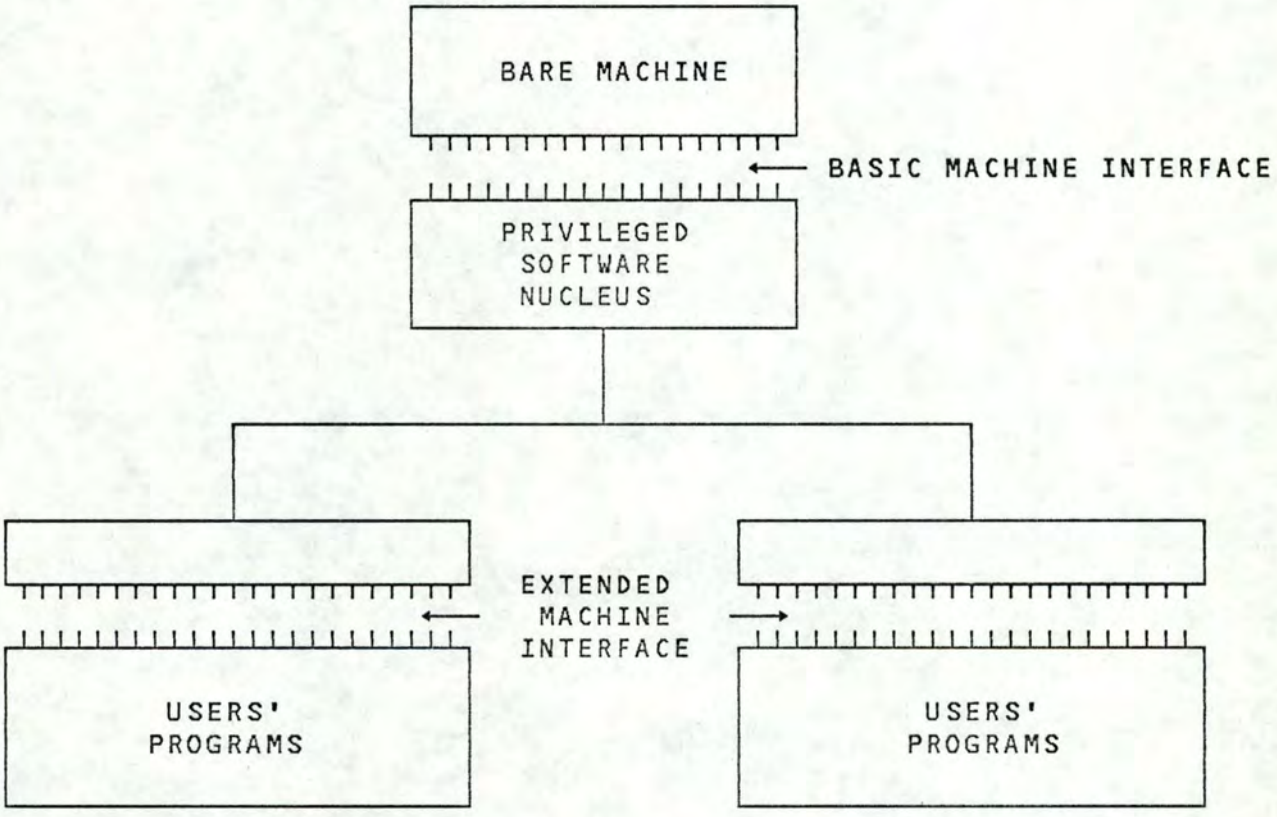


Fig. 2.1: conventional extended machine

Figure 2.1 illustrates the conventional dual state architecture which is responsible for the problems that were cited previously. As can be seen, this system contains only one basic machine interface and is able to execute only one privileged software nucleus. On the other hand, it is able to undergo several extended machine interfaces and therefore several user's programs.

The idea of the virtual machines is to build a particular nucleus which can provide several copies of a basic machine interface instead of several extended machine interfaces.

This would solve all the problems of the third generation architecture mentioned before. As illustrated in figure 2.2, this particular nucleus is known as a Virtual Machine Monitor or VMM. This one provides several duplicates of the bare machine known as virtual machines. Each of the "additional" basic machine interfaces can undergo a conventional privileged software nucleus. This nucleus will be considered by the real machine, through the VMM, as a user's program. Thus, it will be loaded in the user's memory space and executed in a multiprogramming environment.

The VMM realizes the transparency between the nucleus loaded in user's memory and the real machine. That means that the real machine doesn't know the existence of several privileged nuclei running like user's programs and that these nuclei have no way of determining whether they are running on a bare machine or on a virtual machine system.

WARNING.

To avoid confusions, from now on,

- the nucleus running on the virtual machine will be known as "simulated system".
- the nucleus running on the real machine will be known as Virtual Machine Monitor (VMM).

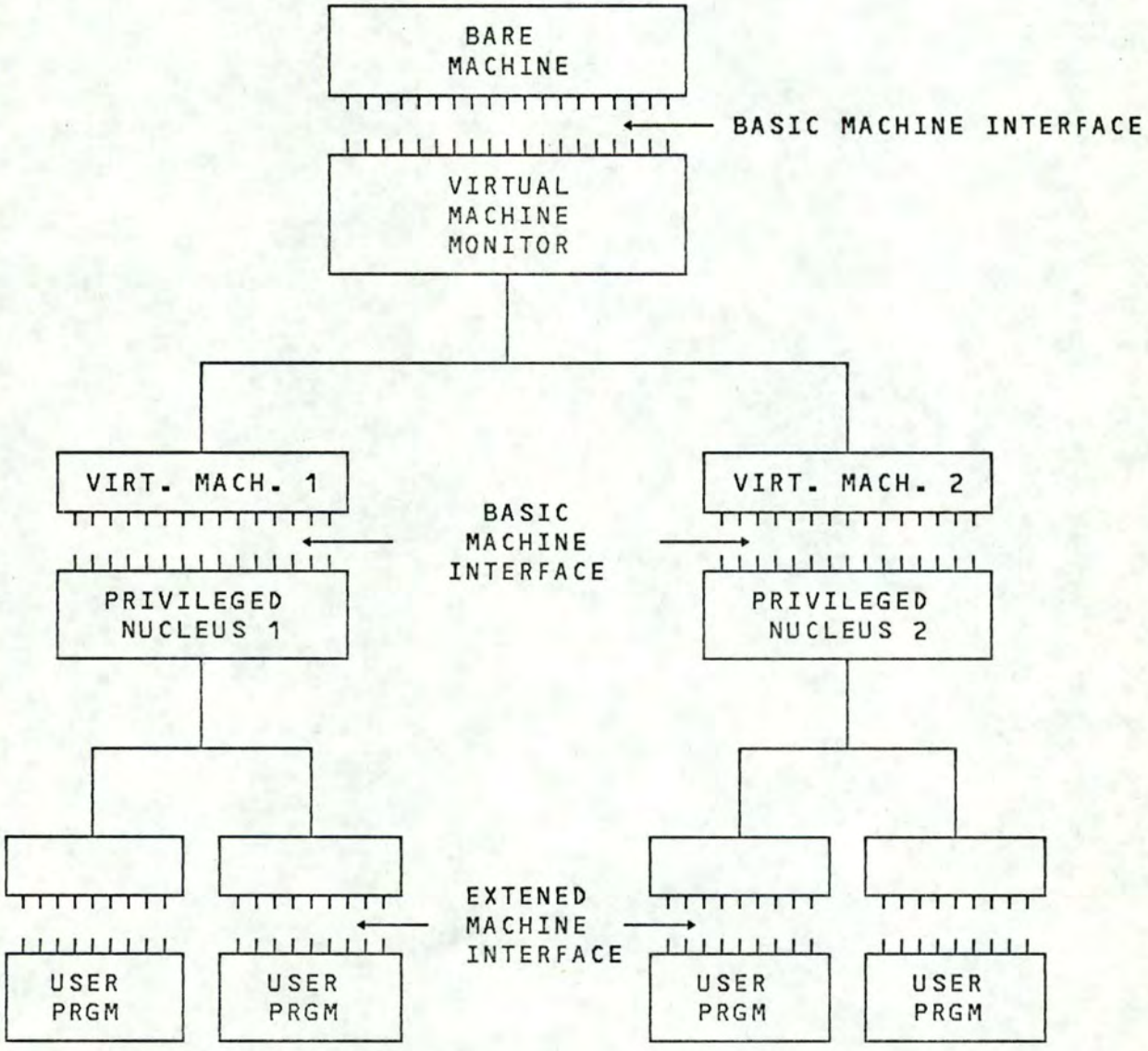


Fig 2.2: Virtual machine configuration.

ANALYSIS OF VIRTUAL MACHINE CONCEPT

3 Analysis of virtual machine concept

3.1 Type of virtual machine. ([2],[3],[4])

The figure 2.2 doesn't imply that the basic machine interface supported by the VMM must be identical to the one supported by the bare machine. When it is the case, the virtual machine is known as virtual machine of type 1. When the two interfaces are different, the virtual machine is known as a virtual machine of type 2. Aside from this comparatively difference, virtual machines of the two types are similar in both structure and functions.

3.1.1 Type 1

For this type of machine, as can be seen on figure 2.6,the VMM runs directly on the bare machine. Thus, the dependance between the VMM and the hardware is very important. Generally, these machines provide the same basic machine interface as the real machine does.

3.1.2 Type 1 bis

This type of machine is a generalisation of the type 1 where a VMM may run a VMM on its interface and therefore be recursive.

3.1.3 type 2

Here, the VMM runs on the extended machine interface provided by the the privileged software nucleus. Thus the VMM has access to all the facilities provided by the extention of the instruction set. That means less dependance between the hardware and the VMM.

3.1.4 Diagrams of the different types

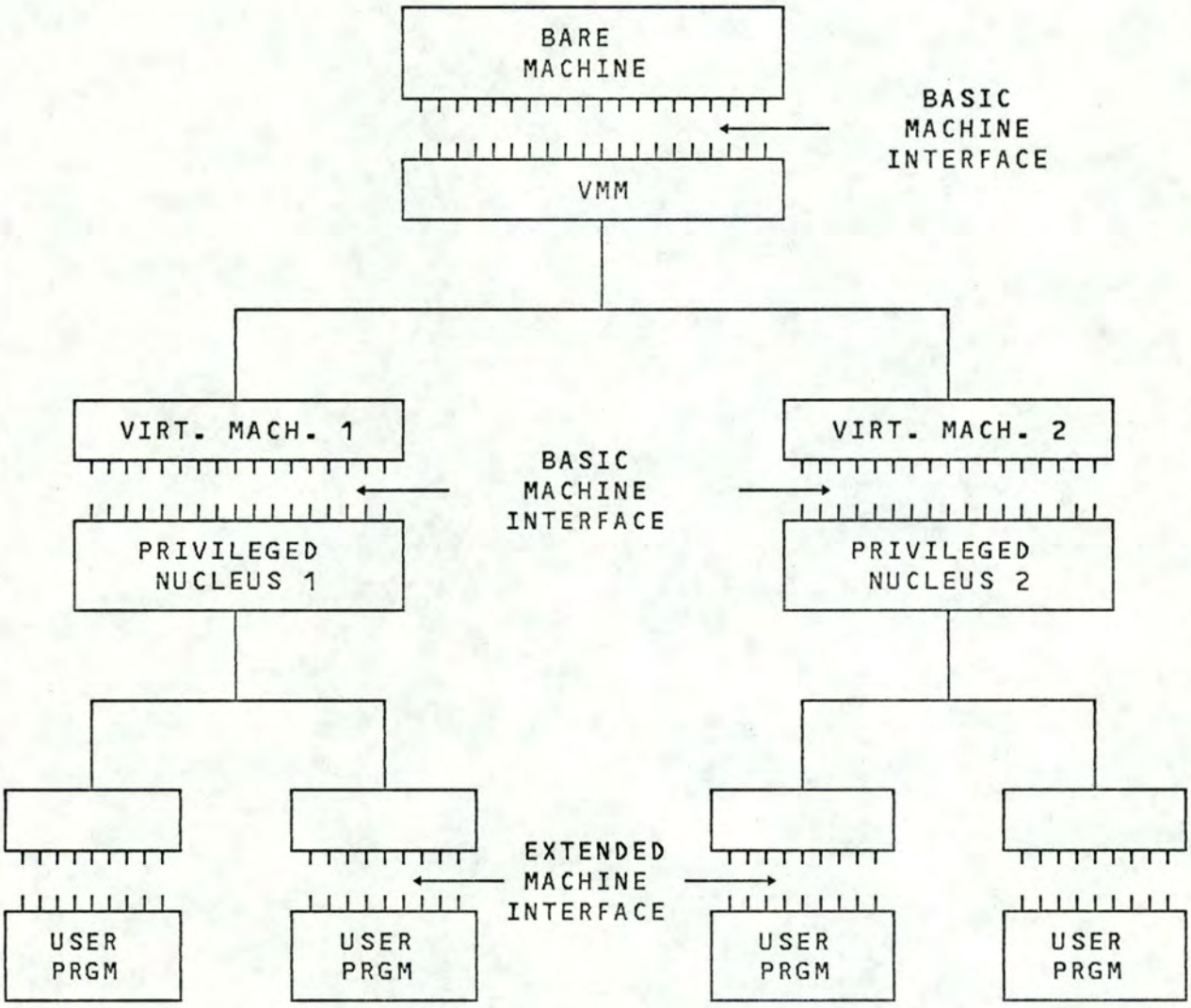


Fig 3.1: virtual machine of type 1.

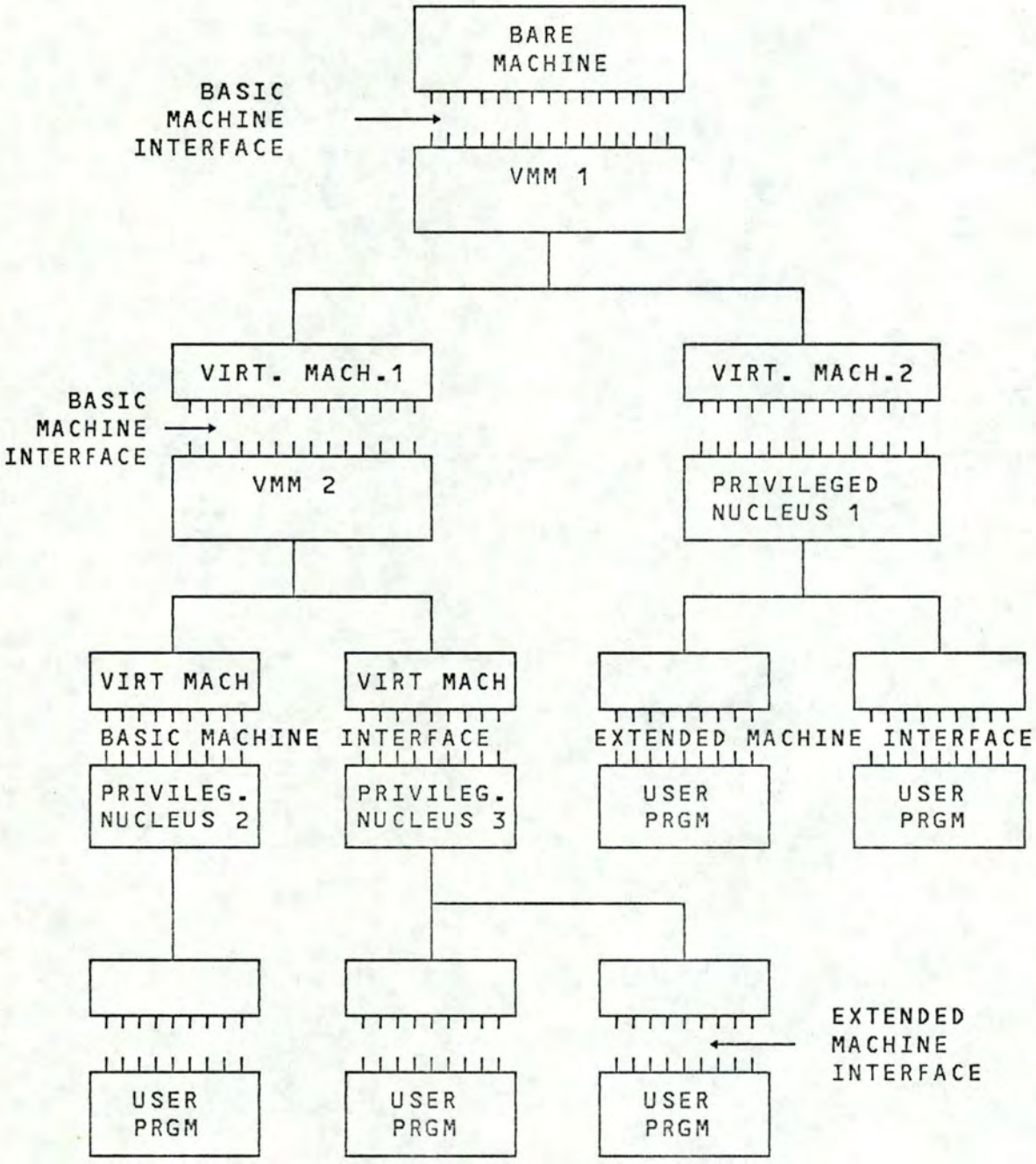


Fig 3.2: virtual machine of type 1 bis.

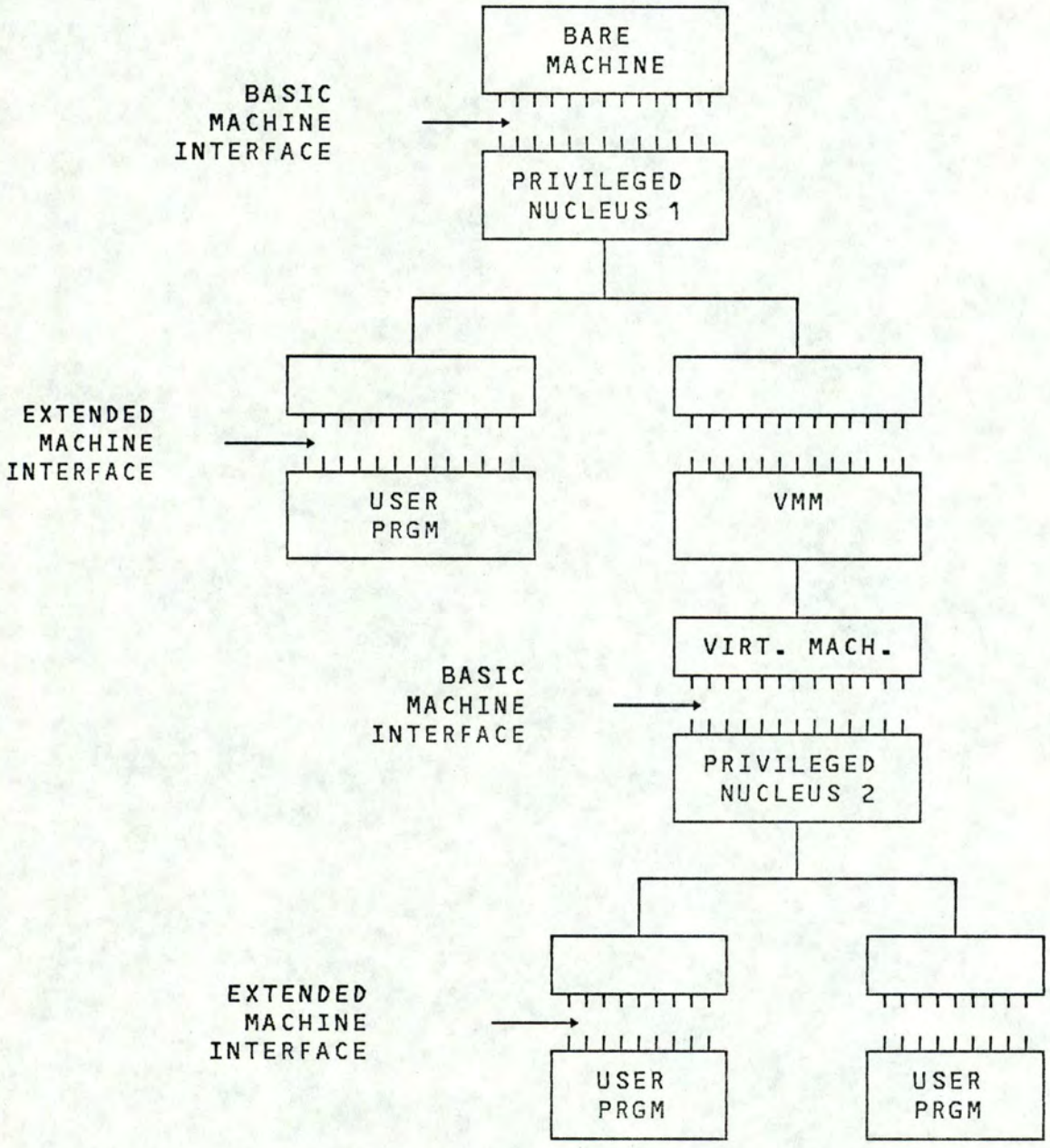


Fig 3.3: virtual machine of type 2

3.1.5 Comparison between type 1 and type 2

3.1.5.1 Performance

On the point of vue of performance, the type 1 is superior because of the fact that the VMM runs directly on the hardware of the real machine and can thus simulate the privileged instructions by means of the microcode. In the type 2, all the privileged instructions are simulated by software and thus require the execution of 250 to 400 supplementary instructions.

3.1.5.2 Resources

For both types, all the software resources are supplied by the different virtual machines, whereas the hardware resources are supplied by the VMM which also realizes the time sharing between the different virtual machine.

3.1.5.3 Cost of implementation

Virtual machines of type 2 offer some implementation advantages: indeed the VMM which runs on the extended machine interface can take profit of the extended machine's instruction repertoire and can be, therefore, easier to construct than VMMs running directly on a bare machine.

3.2 Major features of virtual machine concept

In the way of working, the differences between a conventional computer system and a virtual machine system are generally located in the areas of privileged instructions execution, virtual addressing and I/O's performing.

3.2.1 Simulation of privileged instructions. ([4],[6],[8])

The most significant aspect of virtual machine monitor is the way in which programs are executed. The VMM doesn't execute them instruction by instruction but allow them to run directly on the host system for much of the time. However, the VMM will trap the critical instructions to treat them interpretively in order to insure the integrity of the system.

In the third generation architecture, an attempt to execute a privileged instruction in non privileged mode causes a interrupt which leads to an abnormal termination of the running program.

For the virtual machine, the principle is quite different. As seen before, the privileged software nucleus running under the VMM is considered by the host system as a user program. This nucleus, as all the operating systems, contains a lot of privileged instructions and thus will causes a lot of interrupts. These interrupts work as triggers, for the VMM, to simulate the privileged instructions. In fact, an interrupt causes a change of the host system state from the user to the system mode. The analysis of the interrupt by the system will lead to the conclusion that it is an attempt by the simulated system to execute a privileged instruction. Then, the VMM receives the control in the privileged state in order to simulate, by means of its routines, the privileged instruction wanted by the simulated system. After the simulation, the control is returned to the simulated system and the state of the host system is shifted to the user's one. This mechanism is illustrated in figure 3.4.

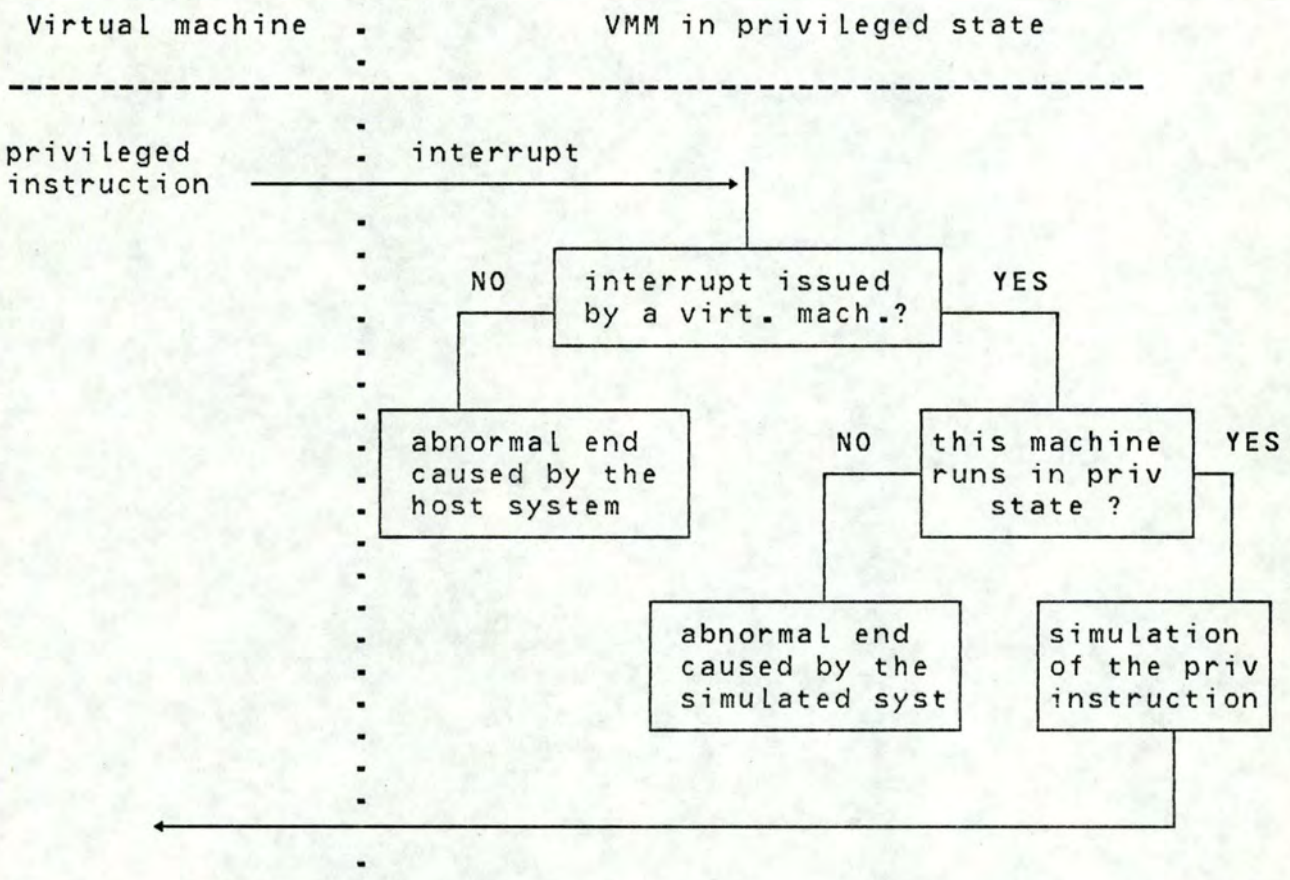


Fig 3.4: simulation of privileged instructions.

3.2.2 Double virtual addressing. ([4],[6])

All the modern operating systems use virtual memory and, as the virtual machine runs under a simulated privileged software nucleus which also use virtual memory, the computing system works with three levels of addressing:

- Level 1: real address of the host system.

- Level 2: real address of the simulated system.
  These addresses are virtual for the host system because the simulated system is contained in its user memory.
  These addresses are called "simple virtual addresses".

- Level 3: virtual address of the simulated system. These addresses are called "double virtual addresses" because they need two translations to become real addresses for the host system.

The address-mapping schematic diagram and associated tables are illustrated in figure 3.5.

In order to execute programs, the central processing unit (CPU) must access instructions and data by means of addresses of the first level. Thus, all the virtual addresses (double and simple) must be translated into real addresses. This double translation would degrade too much the performance of the system if it must be applied each time the central processing unit (CPU) must access data or intructions in central memory. To avoid this, the system must be able to translate the third level directly into the first. In order to achieve this, it exists two methods: the dual paging method and the V=D(M) method. Both are dynamical.

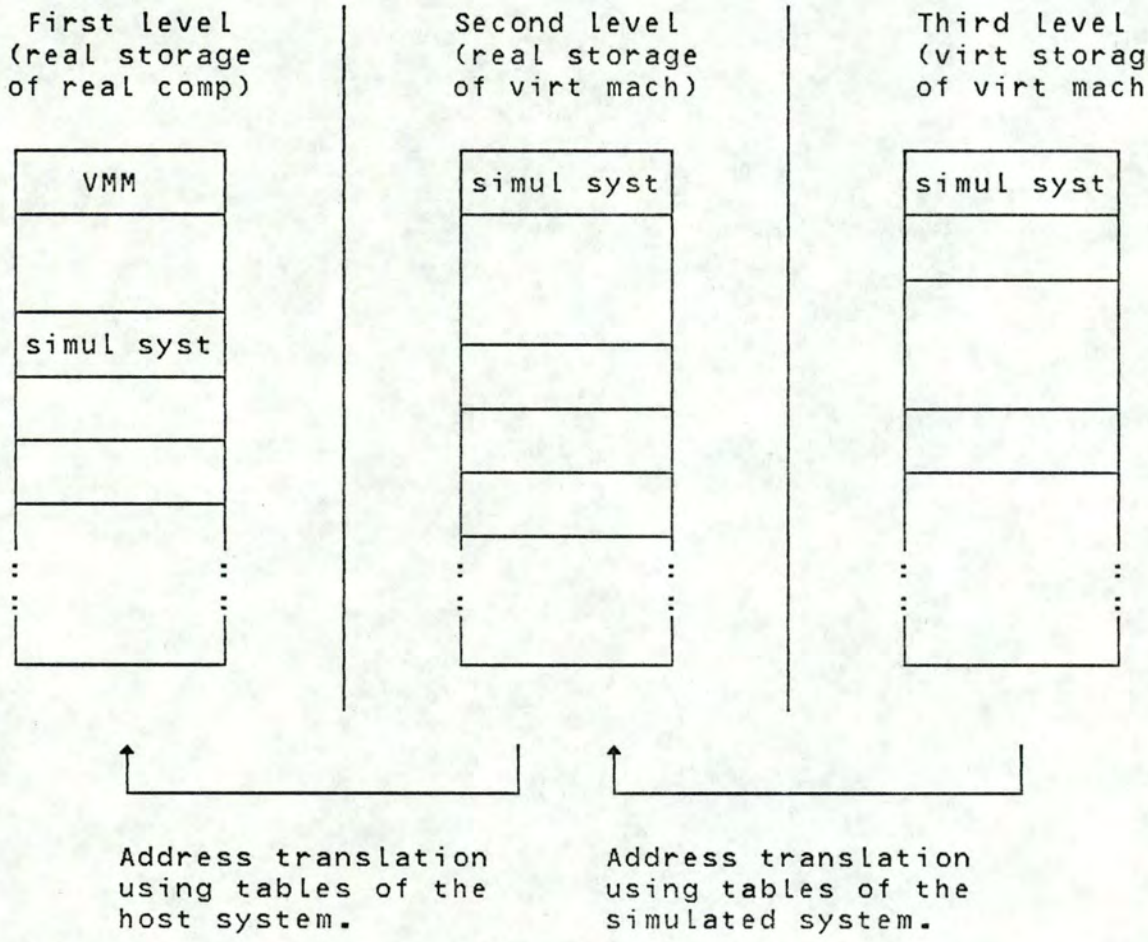
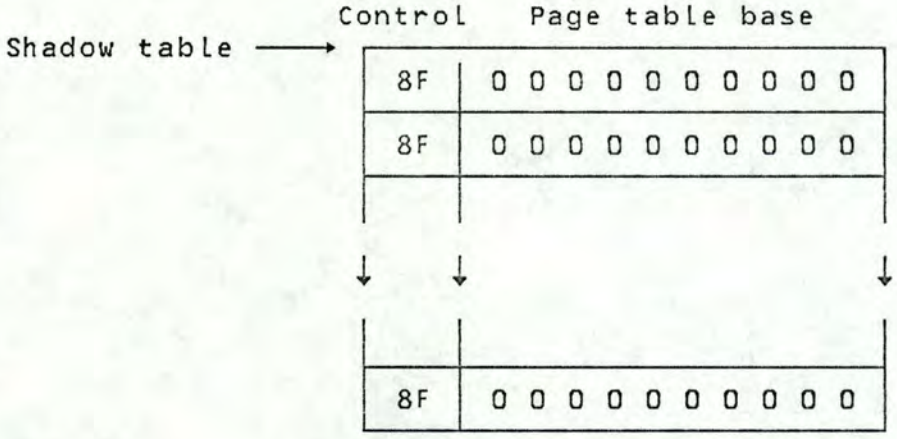


Fig. 3.5: schematic diagram of address translation

3.2.2.1 Dual paging method

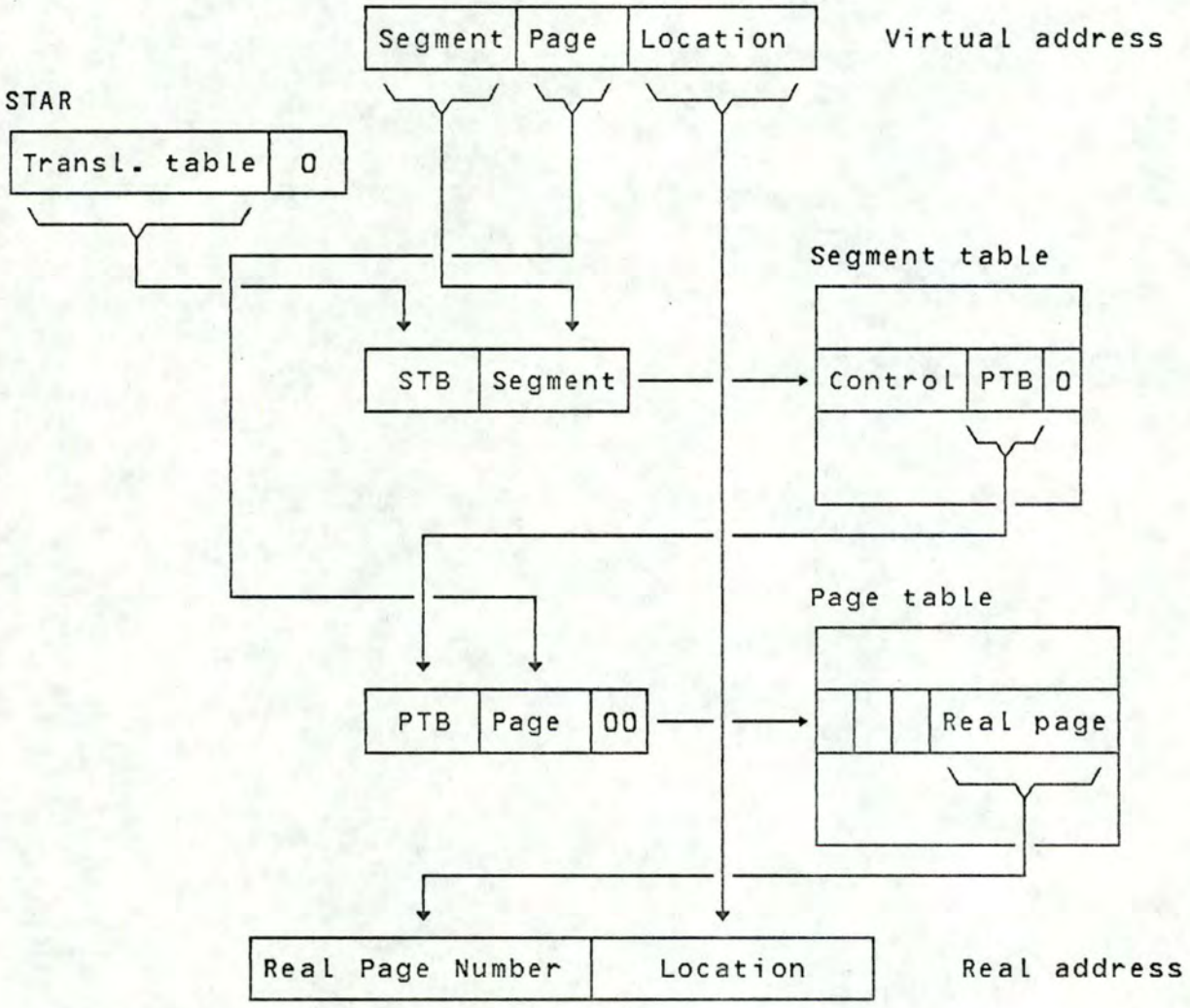
This method uses the classical dynamical address translation feature of the host system illustrated in figure 3.6. Each virtual address may be decomposed in 3 fields: a segment number, a page number within the segment and a location within the page. The segment number determines an entry in the segment table which contains the address of the page table corresponding to the segment. The page table is accessed and the page number determines an entry in this table. This entry gives the real page address where the correct page can be found in real memory.

For the double virtual address translations, a special segment table (called the "shadow table") is built during the initial program loading (IPL) of the simulated system. This table is build as follows:



1. The translation table is initialised with X'8F' in the first byte of each segment table entry meaning that all the segments exist but are not in main memory.

2. When the control is given to the simulated system, the hardware register STAR is loaded with the address of the shadow table.
   At the beginning of the simulated system execution, the first double virtual address will give a paging queue interrupt since the shadow table is empty. It is the paging queue simulation which completes the shadow table by the way illustrated in figure 3.7.



STAR segment table address register
STB segment table base
PTB page table base

Fig 3.6: dynamical address translation feature

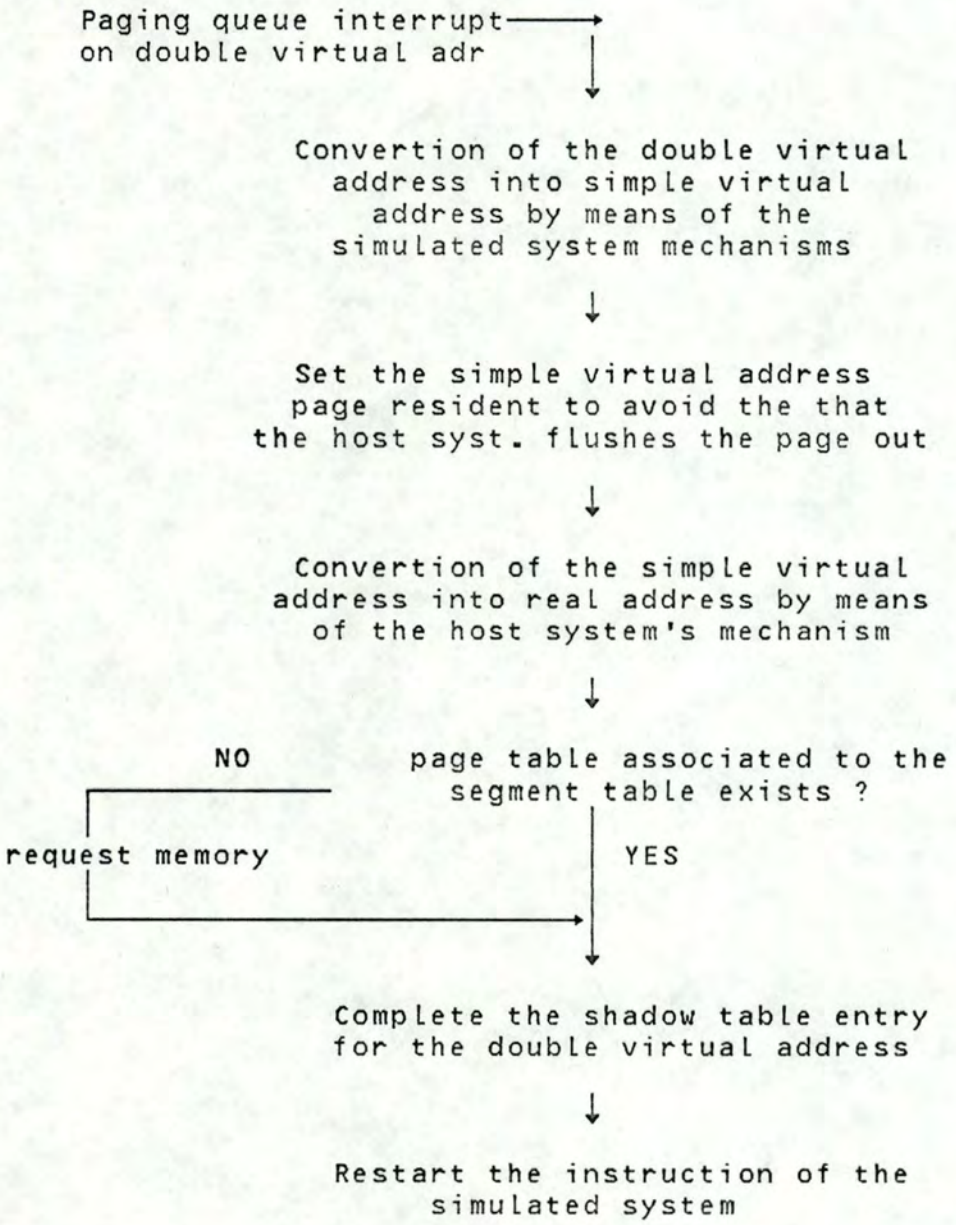

Paging queue interrupt
on double virtual adr
Convertion of the double virtual address into simple virtual address by means of the simulated system mechanisms
Set the simple virtual address page resident to avoid the that the host syst. flushes the page out
Convertion of the simple virtual address into real address by means of the host system's mechanism
page table associated to the segment table exists ?
NO
request memory
YES
Complete the shadow table entry for the double virtual address
Restart the instruction of the simulated system

Figure 3.7.

3.2.2.2 V=D(M) method

The V=D(M) method is intended to reduce overheads (associated with the generation and the management of the shadow table) and to remove the shadow table area. The VMM gives each virtual machine a contiguous real memory area of the real machine while the real storage of the virtual machine is controlled in collaboration by the simulated system and the VMM.

In this method, the shadow table is removed.A contiguous area (from n to m) is allocated to a virtual machine (the length of the simulated system is l), this is illustrated in the figure 3.8. The translation of the third level to the first one is performed as follows:

- the third level address is translate into a second level address by means of the tables of the simulated system.

- if the second level address is $< l$
  then first level address = second level address + m

- when the second level address is $\geq m+l$
  then first level address = second level address

This translation is made each time the CPU must access instructions or data by means of a double virtual address. If it must access instructions or data by means of a simple virtual address, only the second part of the translation explained above is performed.

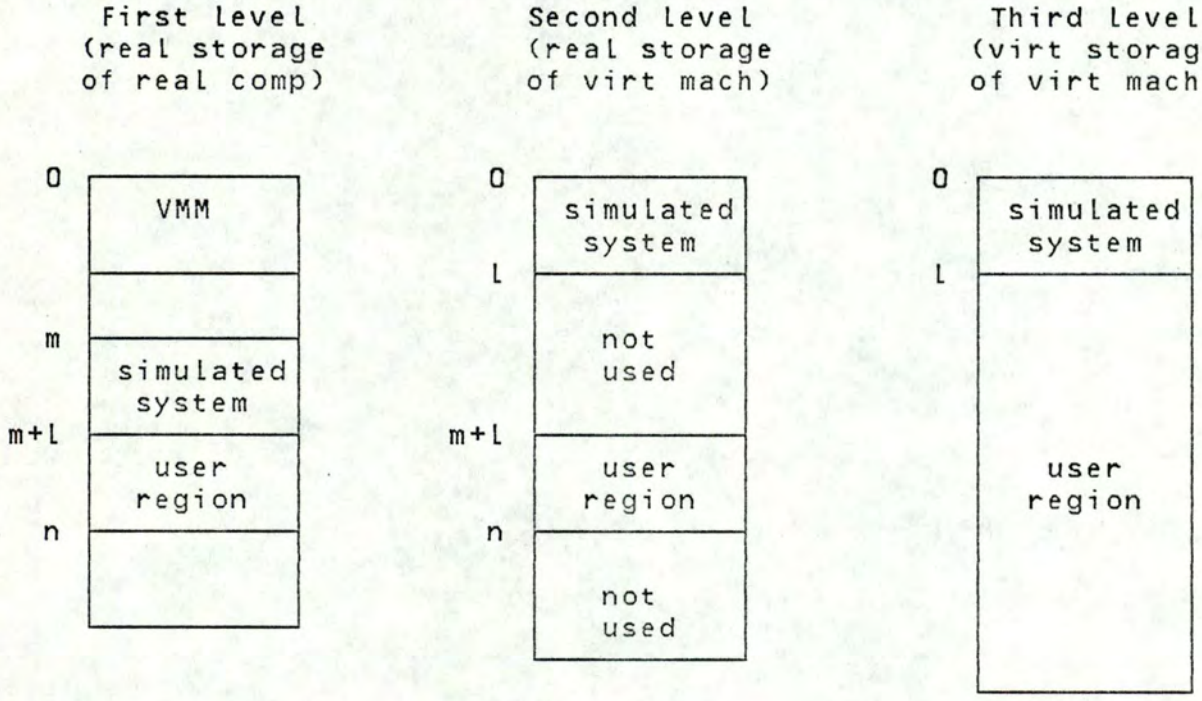


Figure 3.8: address translation for V=D(M) method

3.2.3 I/O simulation. ([2],[6])

Since I/O instructions are privileged, an attempt to execute I/O by software on a virtual machine causes an interrupt trapped by the VMM.

At this point, the VMM is able to translate device and memory addresses before issuing an I/O instruction on behalf of the virtual machine. When I/O completion interrupt returns to the VMM, it is reflected back to the appropriate virtual machine.

The translation of an I/O instruction uses tables built at the initialisation of the virtual machine. Special commands are used to define virtual devices attached to a virtual machine and their real counterparts. All those informations are stored in tables. These ones indicate not only the existance of each I/O element but also the status of the element (e.g. busy or free) and the real hardware component to which it corresponds.

Thus, when a virtual machine issues an I/O instruction, the VMM must first determine that the I/O address is valid in the virtual machine's I/O structure and that the element composing the virtual I/O path (channel, control unit, device) are free. The VMM must then mark the virtual path busy and build an equivalent I/O task for the real hardware. The real path may, of course, be busy, and if so, the task must be deferred until the real path becomes free. Then, the VMM can issue the real I/O instruction corresponding to the virtual one. When the I/O task is completed, the VMM must reflect this fact in the tables describing the virtual machine's I/O structure and simulate the interrupt (end of I/O), including the updating of the virtual machine's channel status word.

A side benefit of the VMM software intervention is the ability to map I/O requests for a device into another or to provide a virtual machine with special devices which have no real counterpart.

3.3 Formal requirements for virtualization ([7])

Virtual machine systems have been implemented on a limited number of third generation computer systems. From previous empirical studies, it is known that certain third generation computer systems cannot support a virtual machine system. J. Popeck and R. Goldberg used formal techniques to derive precise sufficient conditions to test whether an architecture can support virtual machines or not. Those conditions are expressed through three theorems.

Before introducing these theorems, it would be appropriate to define some notions that will be used.

Privileged instruction: an instruction is privileged if, and only if, its execution in the privileged state doesn't bring out an interrupt, though it will in the non privileged state.

Sensitive instruction: we can define two types of sensitive instruction, the control and the behaviour.

Control Sensitive instruction: an instruction is control sensitive if its execution attempts to change the amount of resources available, or affects the processor mode without accessing informations contained in the memory.
example: - on the PDP-10, JRST 1 which is a return to user mode.

Behaviour sensitive instruction: an instruction is behaviour sensitive if the effect of its execution depends on the value of the relocation-bound register, i.e. upon its location in real memory or upon the current mode.
example: - for SIEMENS, LBF (load bit field) which loads bits in different registers following the current mode.
- for DEC on the PDP-11/45, MFPI (move from previous instruction space), this instruction forms its effective address from informations which depend on the current mode.

3.3.1 Basic condition

Theorem 1: For any conventional third generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the privileged instructions.

This theorem gives a very easy and sufficient condition to guarantee the virtualization of a computer of the third generation architecture.

The necessity for the set of sensitive instructions to be a subset of the privileged instructions is that the VMM must trap these instructions in order to be able to simulate them. For example, let's take a sensitive instruction depending on the current mode. If it won't be a privileged instruction, it would be executed directly by the real machine. As the simulated system is considered by the real machine as a user program, the current mode would always be the user's one and as the sensitive instruction could be issued by the simulated system, it won't be correctly executed. So, all the sensitive instructions are to be trapped and thus must belong to the set of privileged instructions.

3.3.2 Recursive virtualization condition

Theorem 2: A conventional third generation computer is recursively virtualizable if it is virtualizable and if a VMM without any timing dependencies can be constructed for it.

This theorem is nearly evident as a VMM provides by definition an environment in which running programs will have identical effects to the ones shown when running on the bare machine. Thus, it is possible to build a VMM running on the basic interface provided by another VMM as it is possible to build a VMM running on a bare machine (cfr. theorem 1). The only constraint is the time dependency. If the VMMs inbeded are time dependend, the recursive virtualization won't be possible because of the comsuption of time made by each of them. Another constraint, acting as a limit on the depth of recursion (number of nested VMMs), is the space available in the machine (since each VMM takes up place, which is quite reasonnable).

3.3.3 Hybrid virtual machines condition

Theorem 3: A hybrid virtual machine monitor can be constructed for any conventional third generation machine in which the set of user sensitive instructions is a subset of the privileged instructions.

In order to discuss this theorem, it is first necessary to specify what is an hybrid virtual machine monitor (HVMM). Its structure is almost identical to a VMM, but more instructions are interpreted rather than being directly executed.

So, the only instructions which can cause problems are the user sensitive instructions (sensitive instructions executed in the user mode), that is why they must belong to the set of privileged instructions in order to be trapped and simulated by the HVMM to insure their correct execution.

Thus, as more insructions are interpreted , a HVMM is less efficient than a VMM. But, as it exists very few third generation architecture which are virtualizable, a HVMM is more actual third generation architecture qualify. For example, the PDP-10 can host a HVMM, although it cannot host a VMM.

IMPLICATIONS FOR THE COMPUTING SYSTEM

4 Implications for the computing system

We are now going to examine the major implications of virtual machine organization for the computing system. These implications are generally knowm as an increase of integrity, a degradation of performance and a special way of sharing data and services.

4.1 Integrity

Operating system integrity may be said to exist when an operating system functions correctly under all circumstances. It is helpfull for better understanding to divide the concept of integrity into three related concepts: reliability, security and availability.

By reliability, we mean the ability of the operating system to continue to supply usefull service in spite of all abnormal software conditions, whether accidental or malicious. That is, we expect the operating system to be able to prevent "crashes".

By security, we mean the ability of the operating system to maintain control of the system resources and thereby prevent users from accidentally or maliciously accessing or modifying unauthorized information.

By availability, we mean the fraction of time that a system is available for operation [20].

4.1.1 Reliability and security ([14],[15],[16],[17])

4.1.1.1 Hierarchical approach to system integrity by using virtual machines

There has been considerable research and numerous attempts to develop "perfect" software ranging from hiring clever programmers, to having every program proofread by two or three programmers, to formal theorem proving. None of these approches have been completely successful for projects as large as a general purpose operating system. Under these circumstances, there are at least two things that can be done: try to develop as much security and reliability as possible, and minimize the impact of a malfunction. Numerous computer scientists have observed that it is possible to simplify the design of an operating system and improve its reliability and security by a careful decomposition, separating the most critical functions from the successively less critical functions as well as separating system-wide functions from user-related functions. This approch has been called "hierarchical modularity".

Figure 4.1 illustrates a conventional two-level operating system with the coexistence of multiple programs. Such a system is susceptible to a security violation if a single hardware or software failure were to occur. One factor contributing to the difficulty of validation entire operating system is that user programs interface is realized through hundreds of parameterized entries (supervisor calls, program interruptions, I/O requests, I/O interruptions, etc...). There is presently no way to systematically validate the correct functioning of the operating system for all possible parameters for all entries. In fact, most systems tend to be highly vunerable to invalid parameters.

Referring again to figure 4.1, we can see some of the factors contributing to the problem. In order to provide sufficient functionality to be effective for a large and heterogeneous collection of user programs and application subsystems, the operating system must be quite comprehensive and, thus, more vulnerable to error. Furthermore, as depicted in figure 4.1, there is not more protection between programs of different application subsystems (e.g. P11 and P21) than between programs of a single application subsystem (e.g. P11 and P12). The reliability and security of such conventional operating systems are sufficiently weak that the military has strict regulations that appear to forbid the use of same information system for both "secret" and "top secret" use, even though using separate systems is more costly.

Figure 4.2 illustrates the virtual machine approach to a physically shared system. This structure has numerous advantages. If we define Pr(Prgm) to be the probability that a given run of a program Prgm will cause a security violation to occur, equations (1) and (2) below are expected to hold:

(1) $Pr(Prgm \mid OS(n)) < Pr(Prgm \mid OS(m))$ for $n<m$

(2) $Pr(OS \mid VMM(k)) < Pr(Prgm \mid OS(m))$ for $k<m$

with - OS(i) refering to a conventional two-level operating system designed to support i user programs

- VMM(i) refering to a virtual machine monitor designed to support i virtual machines

- $Pr(Prgm \mid OS(n))$ = probability that a given run of the program Prgm under the OS(n) will cause a security violation.

Explanation of (1) and (2).

(1) - The probability of system failure tends to increase with the load on the operating system (i.e. the number of requests, the variety of functions, the frequency of request, etc...). In particular, a monoprogramming system, OS(1), tends to be much simpler and more reliable than a multiprogramming system. Furthermore, the m-degree multiprogramming system often requires intricate alterations to support the special needs of the m users, especially if m is large.

(2) - The operating system, OS, on a particular virtual machine has the same relationship to a VMM(k) as a user program , Prgm, has to a conventional multiprogramming operating system, OS(n). In accordance with the same ratio as in equation (1), the smaller the degree of multiprogramming (i.e. k<m), the smaller the probability of a security violation. Furthermore, as a VMM tends to be shorter, simpler, and easier to debugg than a conventional multiprogramming system, even if k=m, the VMM is less error-prone.

If we assume that the events represented by the equations (1) and (2) are independent, we can define the probability of a program Prgm on one virtual machine violating another program on another virtual machine as:

(3) Pr(Prgm | OS(n) | VMM(k)) = Pr(Prgm | OS(n)) * Pr(OS | VMM(k))

Based on the inequalities of equations (1) and (2) and the dependency in equation (3), we arrive at the conclusion:

(4) Pr(Prgm | OS(n) | VMM(k)) << Pr(Prgm | OS(m)) for n,k<m

Pr(Prgm | OS(n) | VMM(k)) is the probability of the simultaneous security failure of Prgm's operating system and the virtual machine monitor. If a single operating system fails, the VMM isolates this failure from the other virtual machines. If the VMM fails, it exposes information of other virtual machines to the operating system of one virtual machine. But, if this operating system functions correctly, it won't take advantages of the security breach. This assumes that the designers of the individual operating system are not in collusion with malicious users, which seems to be a reasonable hypothesis.

We are here particulary concerned about overall security and reliability, that is, the probability of a security failure due to any program in the system. This situation can be computed by:

(5) Pr(Prgm11,Prgm12,....,Prgm33)
= Pr(Prgm11) * (1-Pr(prgm12)) * ..... * (1-Pr(Prgm33))
+ (1-Pr(Prgm11)) * Pr(Prgm12) * ..... * (1-Pr(Prgm33))
+ .....
+ (1-Pr(Prgm11)) * (1-Pr(Prgm12)) * ..... * Pr(Prgm33)

By merging equations (4) and (5) we can conclude that:

(6) Pr(Prgm11,Prgm12,.....,Prgm33 | OS(n) | VMM(k))
<< Pr(Prgm11,Prgm12,.....Prgm33 | OS(m)) for n,k<m

That is, the security and the reliability in a virtual machine environment is very much better than in a conventional multiprogramming operating system. This conclusion, as noted earlier, depends upon the probabilistic independence of security failures. That is what we are going to examine now.

Equations (3) and (4) are based upon the independence of two events: a security failure in Prgm's operating system (OS), and a security failure in the virtual machine monitor (VMM). This hypothesis is reasonable as considering the many sources of accidental security failure. In the case of an attempt to deliberately violate security, the penetrator would usually try to subvert the OS first and then, having taken the control of the OS, attempt to subvert the VMM.

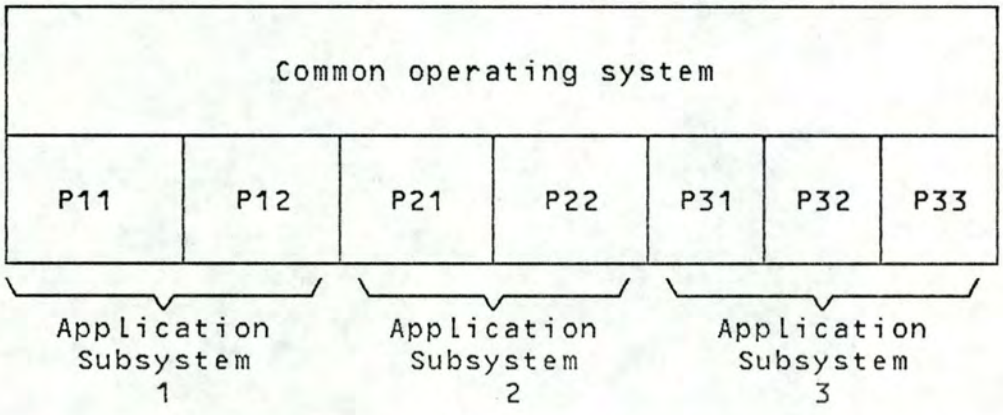


Fig 4.1: Hierarchically structured operating system

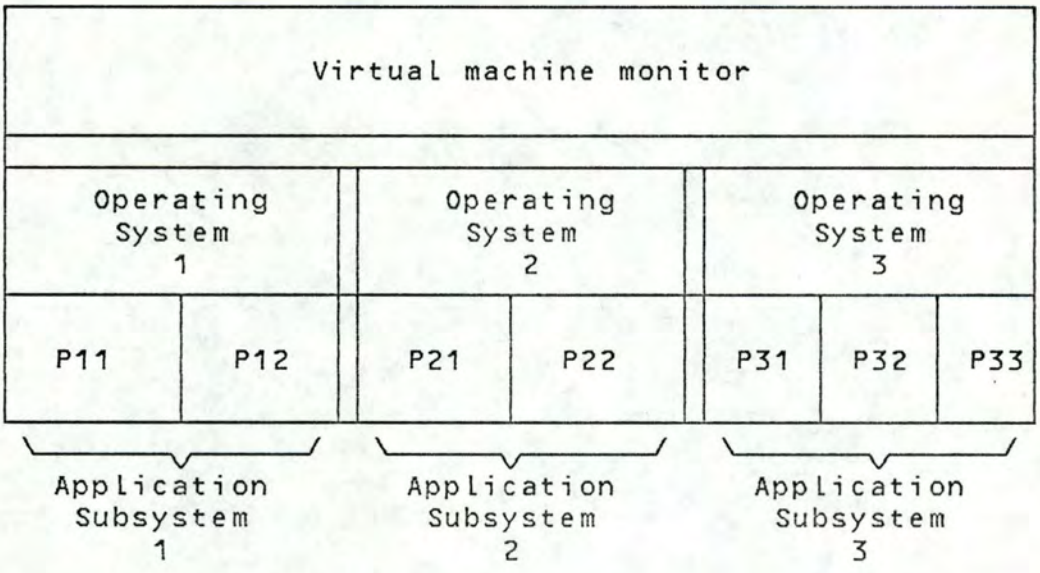


Fig. 4.2: Virtual machine three-level system

4.1.1.2 Redundant security mechanisms

Aside from the benefits of hierarchical approach to computer system integrity, virtual machine concept can minimizes the danger of penetration the OS and the VMM by using redundant security mechanisms.

Let's take for example the store of jewels in a safe. One may think that his jewels are more secure if he stores the first safe in another one. But the foolish man might (so he won't forget) use the same combinaison for both safes. If a burglar figures out how to open the first safe (either accidently or maliciously), he will find it easy to open the inside safe. However, if two different locking mechanisms and combinaisons are used, then the jewels are more secure as the burglar must break the mechanisms of both safes.

Thus, to be the more secure as possible, the OS and the VMM must use different and redundant security mechanisms.

4.1.2 Availability ([14])

Availability has been defined as the fraction of time that a system is available for operation [10]. In the case of multiprogramming systems, this definition has to be modified to reflect the fact that a system may be only partially available for operation. That is, at any given time, a system may not be able to support the maximum level of multiprogramming which could be supported if all components were functionning properly. To reflect this, we define the i-degree of availability (A(i)) as the fraction of time that a system can support i levels of multiprogramming ($0 \leq i \leq n$).

The following section examines the availability of two equivalent systems:

1) OS-n: a multiprogrammed operating system which can support n independent user programs.

2) VMM/OS-1: a virtual machine system which can undergo n copies of a monoprogrammed operating system (OS-1).

Let Av(i) be the i-degree of availability of VMM/OS-1 ($0 \leq i \leq n$) and An(i) be the i-degree of availability of OS-n ($0 \leq i \leq n$).

Note: In the following, we will assume that time between failures and failures recovery time are exponentially distributed.

Analysis of Av(i)

First, note that the VMM/OS-1 configuration will be able to support i user processes ($1 \leq i \leq n$) if and only if the VMM is functioning and exactly i OS-1 are also functioning (the remaining n-i OS-1's are undergoing recovery operations). VMM/OS-1 will not be able to support any user processes if the VMM is functioning and all n OS-1 are down, or if the VMM itself is down.

We have further assume that all the times are exponentially distributed.

Let 1/a = mean time between an OS-1 software failure

1/b = mean recovery time for an OS-1 failure

1/c = mean time between a VMM software failure

1/d = mean recovery time for a VMM failure.

The system can thus be regarded as a continious time Markov process having the following 2n states:
{0,1,2,...,n,0',1',2',...,n'}

state i: i working copies of OS-1 and n-i copies of OS-1 undergoing recovery operation.

state i': system was in state i and a VMM software failure occured.

Let P_i and P_i' be the probabilities that the system is in state i and i' for $0 \leq i \leq n$. The rates of transition are as follows:

state i to i+1 = $(n-i)bP_i$

state i to i-1 = iaP_i

state i to i' = cP_i

state i' to i = dP_i',

The entire system is described in figure 4.3.

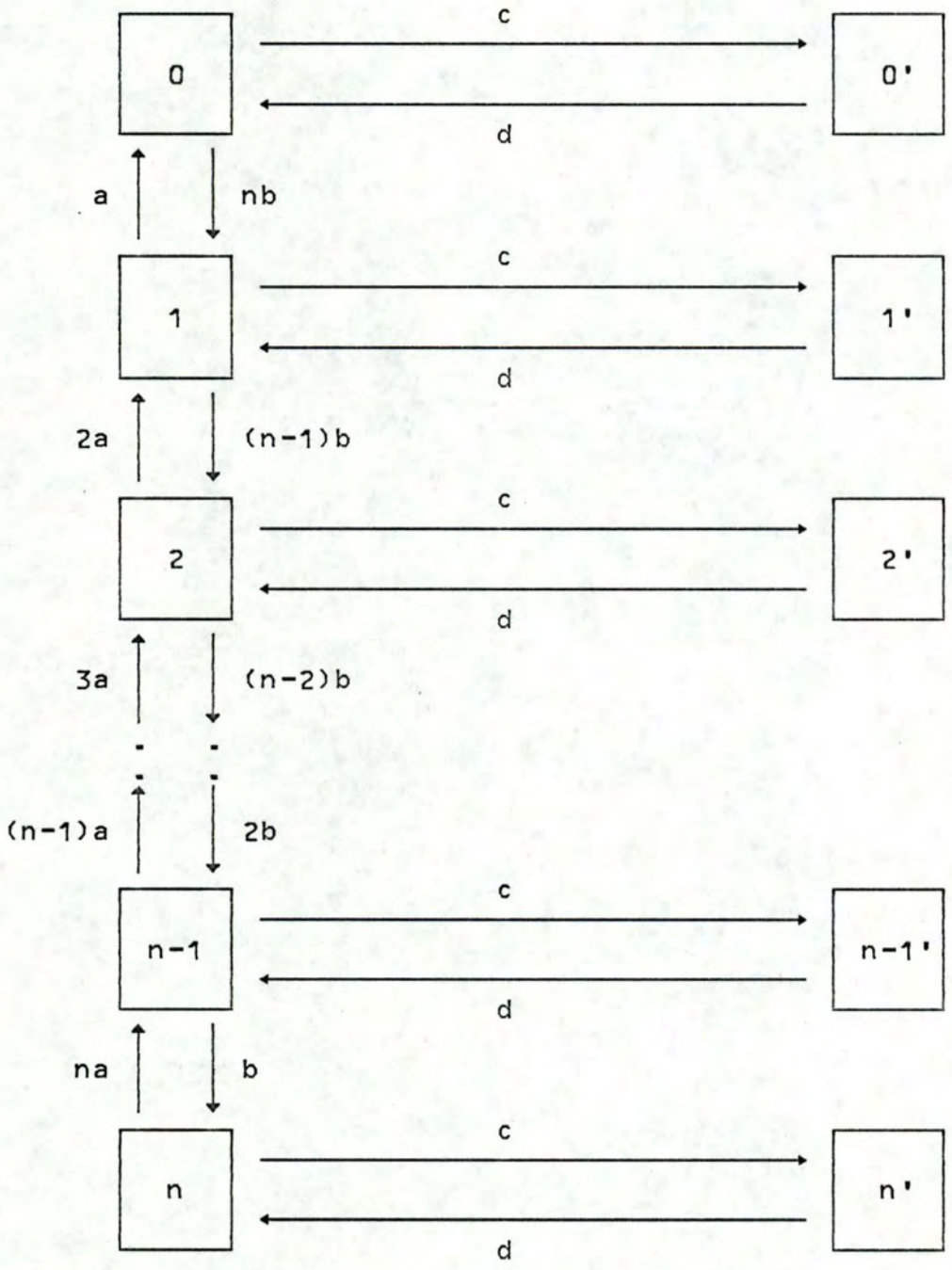


Fig 4.3: state transition diagram.

The balance equations then become:

$$- dP_{i'} = cP_i \qquad (0 \leq i \leq n)$$

$$- (ai + (n-i)b + c)P_i = (n-i+1)bP_{i-1} + (i+1)aP_{i+1} + dP_{i'} \qquad (1 \leq i \leq n-1)$$

$$- (nb+c)P_o = aP_1 + dP_{0'}$$

$$- (na+c)P_n = bP_{n-1} + dP_{n'}$$

The normalizing condition is:

$$- \sum_{i=0}^{n} P_i + \sum_{i'=0}^{n} P_{i'} = 1$$

By solving the above equations, we have:
(the resolution is given in Appendix 1)

$$P_i = \frac{d}{d+c} C_n^i \left[\frac{b}{a+b}\right]^i \left[\frac{a}{a+b}\right]^{n-i} \qquad 0 \leq i \leq n$$

$$P_{i'} = \frac{c}{d} P_i \qquad 0 \leq i \leq n$$

If the VMM is considered as a separate entity, its availability can be expressed as:

$$Av = \frac{d}{c+d}$$

Similary, the availability of each OS-1 can be expressed as:

$$A1 = \frac{b}{a+b}$$

Thus, P_i can be written in terms of Av and A1 as:

$$P_i = Av \; {}^{i}_{n}C \; A1^{i} \left[1-A1\right]^{n-i}$$

The degree of availability of VMM/OS-1 configuration can then be written as:

$$Av1(i) = P_i = Av \; {}^{i}_{n}C \; A1^{i} \left[1-A1\right]^{n-i} \qquad 1 \leq i \leq n$$

$$Av1(0) = P + \sum_{i'=0}^{n} P$$

$$= Av \left[1-A1\right]^{n} + 1 - Av$$

Analysis of An(i)

The OS-n configuration has only two possible states: all n levels are available for operation or no level is available. Assuming the time between two OS-n failures as exponentionally distributed with mean of 1/e and the recovery time with mean 1/f, the degree of availability for OS-n are as follows:

$An(0) = e/e+f$

$An(i) = 0 \qquad 1 \leq i \leq n-1$

$An(n) = f/e+f$

Since OS-n can only be in one of two states, fully available or fully unavailable, the absolute availability will be defined as:

$An = f/e+f$

Utility function

When comparing VMM/OS-1 with OS-n, it is necessary to consider not only the degrees of availability of the two systems but also the relative value associated with each degree. For example, in some cases, it may be absolutely essential to support n levels of multiprogramming at all times; support n-1 or fewer levels may be regarded as totally unacceptable. In such cases, the relative value of the two systems may be assessed by simply comparing Av1(n) and An(n).

In the more general case, there will be a set of value U(1), U(2),..., U(n) associated with the capacity to support 0, 1, 2,..., n levels of multiprogramming. The function U(i) is referred to as a utility function and expresses the relative value of each degree of availability in some particular application. In this case, the overall value of the VMM/OS-1 system may be defined as the expected utility:

$$\sum_{i=0}^{n} U(i)\ Av1(i)$$

Similary, the expected utility of the OS-n system may be defined as:

$$\sum_{i=0}^{n} U(i)\ An(i)$$

Consider any linear utility function defined on {0,1,2,...,n}. That is, assume $U(i) = K\ i/n$ for $K>0$. Since the range of each finite utility function can be transformed into the interval [0,1], these linear utility functions all have the canonical form

$$Ul(i) = i/n \qquad 0 \leq i \leq n$$

Comparing the expected utility of two system using an arbitrary linear utility function is cleary equivalent to comparing expected utility using the function Ul. Utilizing this remark, we can now prove the following theorem.

Theorem: If Av A1 ≥ An and the utlity function is linear, then the expected utility of the VMM/OS-1 system is greater than or equal to the expected utility of the OS-n system.

Proof: the expected utility of the VMM/OS-1 system is:

$$\sum_{i=0}^{n} Av1(i)\ Ul(i) = Av1(0)\ Ul(0) + \sum_{i=1}^{n} Av\ C_n^i\ A1^i \left[1-A1\right]^{n-i} Ul(i)$$

$$= Av1(0)\ 0 + \sum_{i=1}^{n} Av\ C_n^i\ A1^i \left[1-A1\right]^{n-i} i/n$$

$$= Av\ A1 \sum_{i=1}^{n-1} C_{n-1}^{i}\ A1^i \left[1-A1\right]^{n-1-i}$$

$$= Av\ A1$$

The expected utility of the OS-n system is:

$$\sum_{i=0}^{n} An(i)\ Ul(i) = An(0)\ Ul(0) + An(n)\ Ul(n)$$

$$= An(0)\ 0 + An(n)\ 1$$

$$= An$$

Since Av A1 ≥ An, the expected utility of the VMM/OS-1 is greater than or equal to the expected utility of the OS-n system.

To interpret this theorem, note that A1 is the fraction of time that OS-1 is available if it were running on a bare machine and that Av is the fraction of time that the virtual machines supported by the VMM are themselves available. Thus, AvA1 is the fraction of time that each OS-1 running under the VMM is available for user processing. Since the VMM/OS-1 has approximatively ths same complexity as OS-n, one might expect AvA1 to be roughly comparable to An. However, there are a number of reasons to believe that AvA1 will be significantly greater than An.

First, VMM/OS-1 is modular, thus the VMM and OS-1 can be developed independently and checked out individually on a bare machine.

Secondly, OS-1 is less complex than OS-n since OS-1 is a monoprogrammed operating system. Thus, the mean time between two software failures should be substantially greater in OS-1 than in OS-n. In addition, the expected recovery time should be less in OS-1 than in OS-n because of the difference of complexity and the fact that the VMM is able to react to a failure without the intervention of the operator.

Finally, VMM/OS-1 system is potentially more secure and better able to preserve privacy of each user (this aspect is discussed more comprehensively in the preceding section: 4.1.1). Thus, there should be fewer failures caused by successful (or unsuccessful) attempts to "break" the security of the system.

In summary, AvA1 may be well significantly greater than An in many actual systems, and thus VMM/OS-1 system may be distincly be preferable to OS-n when the two systems are evaluated on the basis of a linear utility function.

Corollary: Consider any utility function U for which $U(i) \geq Ul(i)$, $U(0) = 0$ and $U(n) = 1$. If $AvA1 \geq An$, the expected utility of the VMM/OS-1 system is greater than or equal to the expected utility of the OS-n system.

Proof: Since $U(i) \geq Ul(i)$, the expected utility of the VMM/OS-1 system under U is greater than or equal to the expected utility of VMM/OS-1 under Ul. However, The expected utility of OS-n under Ul is equal to the expected utility of OS-n under Ul since $U(0)=Ul(0)=0$ and $U(n)=Ul(n)=1$. Applying the preceding theorem, the corollary follows immediately.

As a consequence of the corollary, the theorem is extended to include a large class of utility functions. Figure 4.4 illustrates a representative family of utility functions to which the results of the theorem is now applicable.

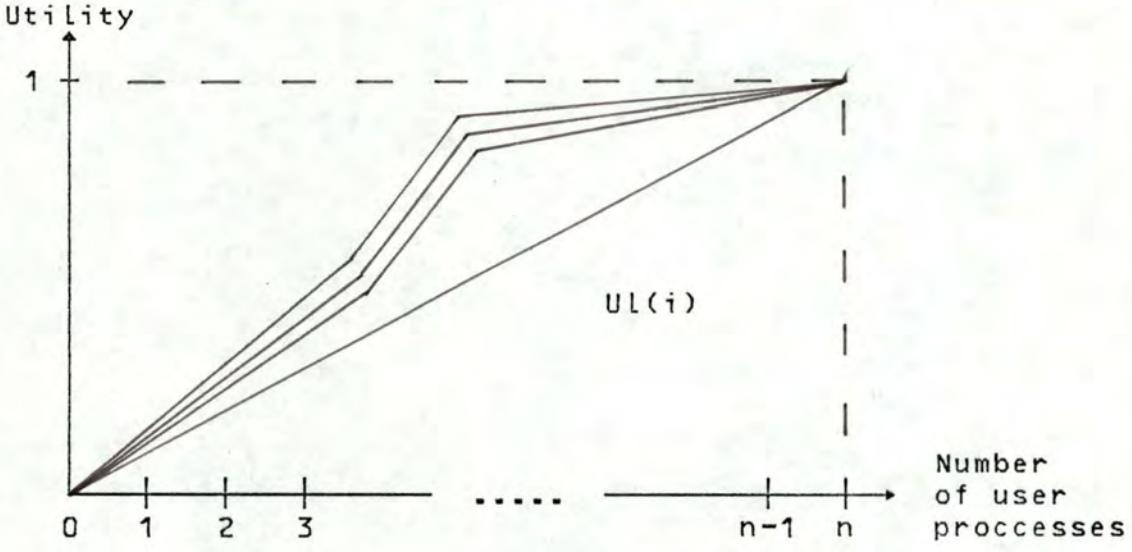


Fig 4.4: Utility functions

4.2 Performance ([18],[19])

Performance degradation is the major desadvantage of virtual machine systems. This degradation is due to the overheads introduced by the VMM. The main sources of overheads are :

- Privileged instructions: VMM spends a large amount of time to simulate the privileged instructions (cfr 3.3.1).

- Maintaining the status of the virtual machines: the scratch pad memory of each virtual machine has to be simulated in order to maintain their virtual processor state. Instead of using the real scratchpad memory of the computer, the VMM uses an area of the central memory to save the state of each virtual machine. When the control is given to a virtual machine by hte VMM, this one loads the real scratchpad memory of the computer with the information contained in the area of the central memory.

- Paging within the virtual machines: if the simulated system works with virtual memory, the VMM has to work with three levels of addressing. Software techniques are used to translate double virtual addresses into simple virtual addresses and finally into real addresses (cfr 3.3.2).

- Addressing a device: all the I/O operations issued by a virtual machine are to be translated in order to be executable (cfr 3.3.3).

All these sources of overheads may be minimized by using microcode. Frequent sequences of instructions such as simulation of privileged instructions, loading and storing of the virtual scratchpad memory, translation of double virtual addresses may be quicked by using microcode instead of using software techniques.

A concrete case of performance degradation is studied in section 6 by means of a practical model.

4.3 Sharing data and services [5]

All time sharing systems offer their users software resources as well as a share of the hardware resources. In the conventionnal third generation architecture, the management of software and hardware resources are integrated and performed by the operating system of the host system. One consequence of the integration is that the sharing of resources among independant users is facilitated.

In a virtual machine system, things are quite different. The software resources are supplied by the simulated system components while the hardware resources are supplied by the VMM. Unfortunately, the VMM "knows" about the other users of the system but is ignorant of the user's file structure; while the simulated system "knows" about the user's file structure but is ignorant of the other users of the system. The division of labor in a virtual machine system, which proved to be an advantage as far as integrity is concerned, proves to be a disadvantage when it comes to the important service of sharing files.

The VMM, which manages the virtual resources of all the users, can be used to implement modes of sharing among virtual machines. This mode of sharing may be thought of as the sharing of hardware realized by the VMM and have been implemented for the main storage (shared segments) and for the auxiliary storage (shared mini-disk). Mini-disks are virtual disks which differ from real disks only in that may have fewer cylinders than a physical disk. They are shared among several users and the sharing may be initiated by the users themselves. Mini-disks are owned by users and the owner may specify passwords to give other users various degrees of access (read-only, read-write, etc...).

But the ability to read from or to write to a mini-disk is limited in value as the contents and the location of the files it contains are not known by the simulated systems. One standard method of resolving this difficulty is to store a directory of the disk contents at a fixed location on the disk itself. Then, the simulated system must first read the directory into its memory before it can access the file on the disk.

APPLICATIONS OF VIRTUAL MACHINES

5 Applications of virtual machines ([2],[4],[5],[6])

There are two main areas of application of virtual machines: the area of development,testing and measurement of operating systems and the area of general purpose, conversational and time-sharing (multi-environment).

5.1 Development, testing and measurement of operating systems

With a conventional third generation architecture computer, the development and testing of a new operating system require the use of a dedicated machine. This makes continued development and modification of the privileged nucleus difficult since system programmers often have to work odd hours (generally during the night) in order to have a dedicated machine.

With a virtual machine system, the problem of the dedicated machine is solved. As it is possible to run several operating systems at the same time, the development and testing of a new operating become as easy as loading and running a user program on a conventional computer.

In addition, as the simulated system is considered through the VMM as a user program, the facilities provided by debbuging and performance tools are available. These advantages should be of great interest for the conceptors of operating systems when thinking to the difficulties met during the conception of a new operating system.

5.2 Multi-environment

Before the appearence of virtual machines, the migration from an old operating system release to a new one caused a big problem, all the programs had to be converted.

Now, with a virtual machine system, it is possible to run concurrently the old and the new release for an extended period of time to alow the users to convert their programs. In addition, when most users programs are finally converted, it is still possible to run the old release for programs which run so infrequently that the conversion is not justified.

Figure 5.1 illustrates how the shift from an old release to a new one can be accomplished using virtual machine techniques. As time advances, the relative percentage of users running under the new release increases till it just remains the permanently unconverted users programs running under the old release.

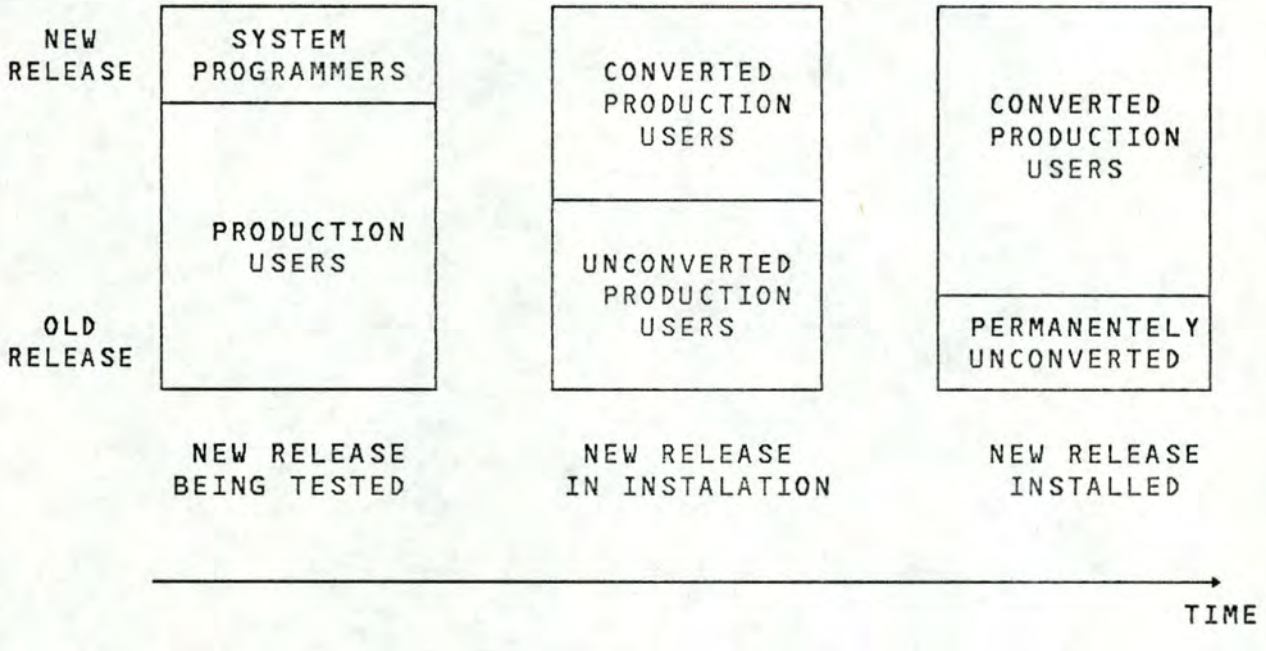


Fig. 5.1: Virtual machine support for multiple releases of an operating system.

PERFORMANCE OF SIEMENS SIM7000

6 Performance of Siemens SIM7000

6.1 SIM7000 of SIEMENS ([21],[22],[23],[24])

6.1.1 Origin of SIM7000

The origin of SIM7000 goes back to the year 1973, when it has been stated that BS2000 would be the most economic and promising operating system. The most important problem which arisen was the partial incompatibility with the BS2000 predecessor: BS1000. A lot of solutions have been studied in order to alleviate the "trauma" of migration; for example: macro solution, conversion system, construction of BS1000/BS2000 interface, etc... There was something that all methods agreed: the efforts made for conversion and compatibility were successful in a certain percentage which is comfortable but not total. This fact has been established in practice, most of the methods neglected the aspect of data compatibility.

The best solution would be the one which could reach a 100% compatibility for both software and data. This total compatibility can be found in virtual machine principles. Virtual machines simulates, by means of software routines, the hardware/software interface, so that an operating system works in a virtual machine just like it would do in a real machine. The total compatibility is reached with virtual machines and even no change in the simulated system is necessary.

The basic goal of SIM7000 was thus to allow the coexistence between a host BS2000 system and one or more simulated BS1000/BS2000 system on one installation and thereby, support the transition from one operating system to another (BS1000 --> BS2000) and the coexistence of two different operating systems (BS2000 version 6 and BS2000 version 7 for instance).

6.1.2 Introduction to SIM7000

SIM7000 allows the simulation of the operating systems BS1000, BS2000 or any other self-loading programs on the extended machine interface provided by BS2000 running on the SIEMENS computers 4004, 7760, 7551, and 7755 central unit.

Peripherical devices may be assigned dynamically to either the simulated system or the host system by the SIM7000 virtual machine. The unit record devices (card reader, puncher, printer) assigned to the simulated system may be mapped to the host BS2000 SAM or ISAM files and thus served via host BS2000 SPOOL functions. The simulated console can be represented by the BS2000 console or by a terminal.

During execution under SIM7000, the simulated system and its user tasks can be tested using the BS2000 interactive debbuging aids (IDA and AID).

Error handling is performed by the simulated system execpt for the machine error category (power failures, machine checks).

6.1.3 Architecture of SIM7000

SIM7000 is a virtual machine monitor of type 2. The VMM is a part of BS2000 and thus can take profit of the already existing management of memory, I/O, and CPU. The figure 6.1 shows how SIM7000 is implemented on the conventional third architecture of the SIEMENS computers.

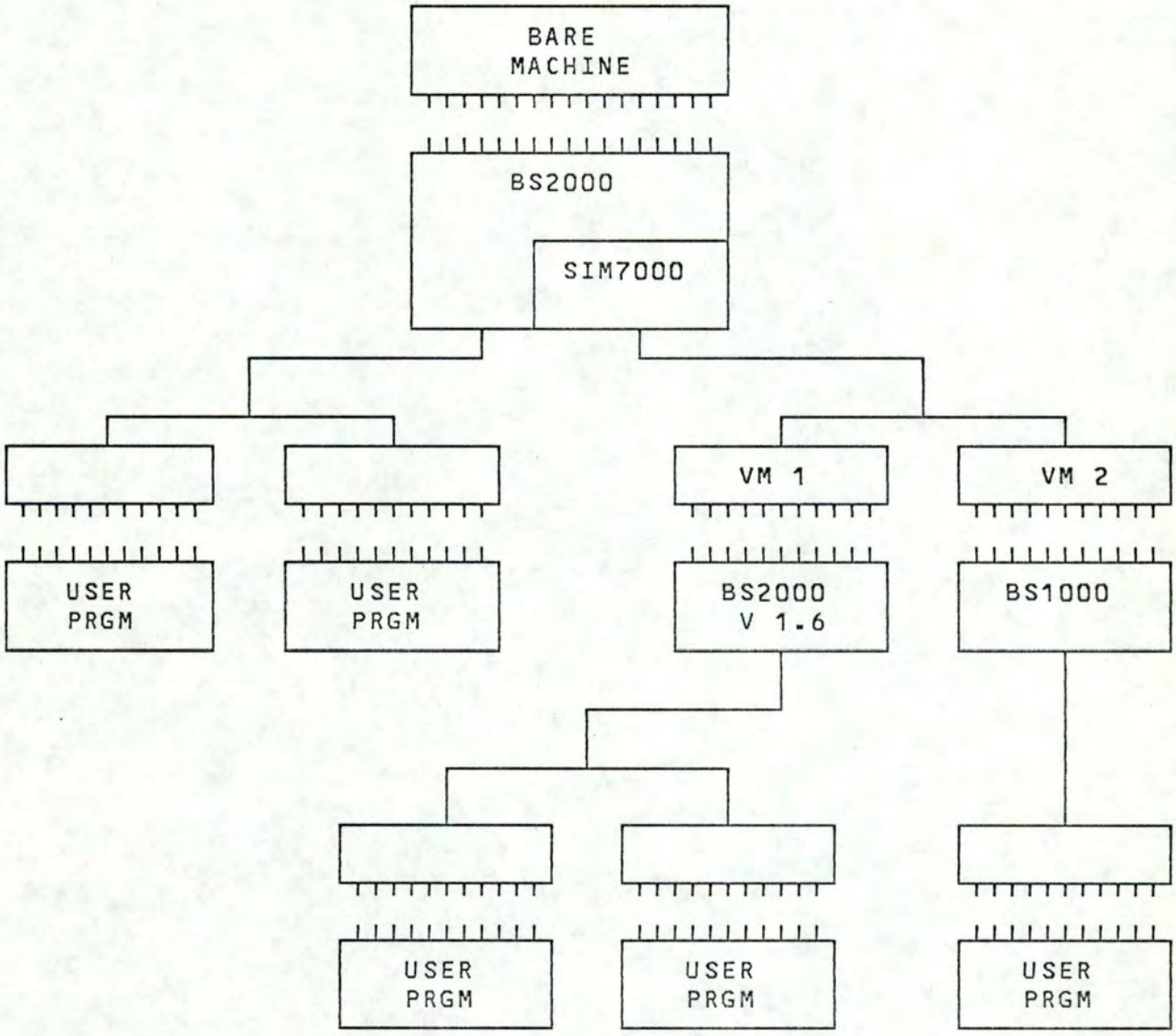


Fig. 6.1: configuration of SIM7000.

6.2 Performance study

6.2.1 Content

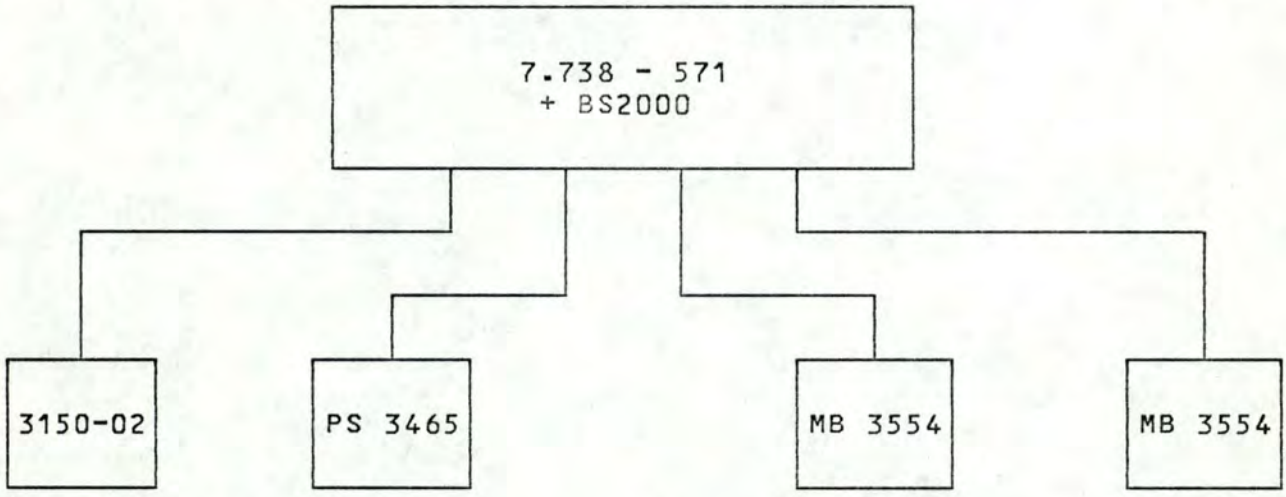
In order to quantify the degradation of performance for SIM7000 of SIEMENS, a special tool has been built. This tool is a self-loading program, that is, a program able to be loaded with IPL procedure and to run on a bare machine (without any operating system). This program measures the execution time of privileged instructions. It will be executed two times. First on a bare machine, without any operating system. This measure will give the real execution time taken by each privileged instruction. Secondely, on the same machine but managed by BS2000 + SIM7000. This measure will give the real execution time taken by the simulation of each privileged instruction. These two measures will be compared to quantify the degradation of performance.

6.2.2 Presentation of the tool

Written in SIEMENS assembler, this tool contains two parts. The first one is the initialisation of the hardware registers and the second one is the measure of the privileged instructions. The complete text of the self-loading program may be found in the annex named "Modules of measurements" joined to this document. Each part of this program will be there largely explained and commented.

6.2.3 Presentation of the hardware

The measurements have been done on the SIEMENS computer exclusively dedicated for tests and located at the SIEMENS center of Namur. The figure 6.2 shows the part of the configuration of this machine used by the self-loading program.



7.738 - 571 (central unit) : 2MB of central memory
CPU of 620 KOPS

3150-02 (card reader) : 1000 cards / minute

PS 3465 (magnetic disk) : capacity of 144 MB
806 KB / second

MB 3554 (magnetic tape) : 1600 bpi
308 KB / second

Fig. 6.2: configuration of the hardware

The first magnetic tape is used by the IPL procedure and contains the object of the self-loading program, the second one is used to measure the time taken by operations of reading and writting on a tape. The magnetic disk and the card reader are also used for measurements.

6.2.4 Different parts of the measurements

Before going on, we must distinghuish two different kinds of privileged instructions that are, for the first one, generally related with the control registers of the computer and the second one is concerned with the input/output privileged instructions. These two kinds of priviliged instructions will be here examined separately because of the evident differences of functionnality.

6.2.4.1 Results of the measurements of the system control instructions

Priv. instr	real time in millisec	simul. time in millisec	simul. - real / real *100
PC	0.01264	0.57718	4466
LSP(1 WORD)	0.00727	0.51742	7017
LSP(16 WORDS)	0.01137	0.69317	5996
LSP(ALL CONTEXT)	0.02486	0.96788	3793
SSP(1 WORD)	0.00722	0.38301	5204
SSP(16 WORDS)	0.01617	0.45828	2734
SSP(ALL CONTEXT)	0.03597	0.88127	2350
LSAL	0.01408	0.45793	3152
SSAL	0.00622	0.36414	5754
STIF	0.00664	0.33256	4908
TSR	0.00715	0.39963	5489
STID	0.00639	0.33600	5158
STNU	0.00582	0.39607	6705
LDWR	0.00621	0.40244	6380
LDHR	0.00624	0.40250	6350
STWR	0.00717	0.40112	5494
STHR	0.00751	0.40178	5249
TDV	0.02246	0.41536	1750
STIO	0.01010	0.34164	3282

Fig. 6.3: table of results.

Looking at the results of the measurements (given in fig. 6.3), we can see that a privileged instruction can take from 17 to 70 times more in a virtual machine system than in a real environment. This could seem huge, but it must be balanced by the fact that all the privileged instructions are only a small part of an operating system and that the normal instructions take the same time in both environment. That's why we have tried to find a ponderation to those measurements.

6.2.4.2 Ponderation

The ponderation must take into account the fact that the privileged instruction are only a small part of an operating system. To quantify this quota, it has been decided to modify the code of the Siemens' VMM, SIM7000. Reffering to section 3.2.1, an attempt to execute a privileged instruction in a virtual machine system causes an interruption. The state of the virtual machine is analysed and if the privileged instruction is authorized, it is simulated by the VMM. Thus, to quantify the number of privileged instructions being executed, we add a counter system in all the simulation routines of the VMM. Thus, while running, the modified VMM will count the number of each type of privileged instruction being executed. But to be complete, we must also count the total number of instructions, privileged or not, that are executed. This has been quite a little bit more difficult. for this, we use the mechanism of the debbuging system of BS2000. This one generates, when set on, an interruption each time an instruction is executed. It is this interruption that allows to trace, instruction by instruction, the execution of a program. We set the debbuging system on, but instead of tracing all the instructions, we simply add one in a counter and bypass the tracing mode. Thus, as the debbuging system generates an interruption for all the instructions, we get the total number of instructions executed by the simulated system. All the counters are given in the fig. 6.4.

Priv. instr	Number of instructions executed	Number(inst) / Total number = ratio
PC	11 232	0.009346
LSP(1 WORD)		
LSP(16 WORDS)	4568	0.003806
LSP(ALL CONTEXT)		
SSP(1 WORD)		
SSP(16 WORDS)	5308	0.004433
SSP(ALL CONTEXT)		
LSAL	0	0
SSAL	0	0
STIF	3917	0.003264
TSR	0	0
STID	2	0.000001
STNU	0	0
LDWR	0	0
LDHR	0	0
STWR	0	0
STHR	0	0
TDV	20	0.000016
STIO	3199	0.002665

with total number of executed instructions = 1 199 256

fig. 6.4: table of ratio.

6.2.4.3 Statistical study

As the ratio (given in fig 6.4) gives the proportion of privileged instructions among all the instructions, we may multiply this one by the percentage of degradation (given in fig. 6.3) to obtain a relative degradation of each privileged instruction. This relative degradation is given in fig 6.5.

Privileged instructions	degradation (in %)	ratio	relative degradation (in %)
PC	4466	0.009346	41.73
LSP(1 WORD)	7017		
LSP(16 WORDS)	5996	0.003806	21.32
LSP(ALL CONTEXT)	3793		
SSP(1 WORD)	5204		
SSP(16 WORDS)	2734	0.004433	15.20
SSP(ALL CONTEXT)	2350		
LSAL	3152	0	0
SSAL	5754	0	0
STIF	4908	0.003264	16.01
TSR	5489	0	0
STID	5158	0.000001	0.01
STNU	6705	0	0
LDWR	6380	0	0
LDHR	6350	0	0
STWR	5494	0	0
STHR	5249	0	0
TDV	1750	0.000016	0.02
STIO	3282	0.002665	8.74

fig. 6.5: table of relative degradation

The relative degradation gives the percentage of degradation for each privileged instruction. For example, we can say that all the PC instructions of the simulated system degrade the performance of about 41 %.

Computing of the statistical data:

$$\text{Mean degradation} = \frac{1}{15} \sum_{i=1}^{15} \text{rel. degradation}_i$$

$$= 6.86\ \%$$

$$\text{total degradation} = \sum_{i=1}^{15} \text{relative degradation}_i$$

$$= 103.03\ \%$$

6.2.4.4 Criticism of the ponderation

The simulated system measured was a BS1000 V1.52. It is one of the versions of BS1000 for which SIM7000 was designed when BS2000 appeared. This version has one main characteristic: it does not use the virtual memory system (that is why some privileged instructions are not used). Execpt this, it is interresting to measure this operating system as it is still used by customers under SIM7000.

One of the possible extension to this work would be to realize the same measures, but for a BS2000 operating system which use the virtual memory system.

6.2.4.5 Conclusion of the privileged instruction measures

The relative degradation of 103.03 % for the privileged instructions is quite acceptable when thinking to the advantages of simulating a BS1000 operating system under a BS2000 + SIM7000 operating system.

6.2.4.6 Results of the input/output instructions

Priv. instr	real time in millisec	simul. time in millisec	$\frac{\text{simul.} - \text{real}}{\text{real}} * 100$
SDV (WRITE 2KB ON TAPE)	7.88772	15.51715	97
SDV (READ 2KB ON TAPE	8.32652	19.19182	94
SDV (WRITE 2KB ON DISK)	16.23138	26.14445	61
SDV (READ 2KB ON DISK	15.98359	26.09564	63
SDV (CONSOLE)	40.67227	324.02337	697
SDV (READ A CARD)	68.98408	84.69609	23
PAGING	22.46381	37.78466	68

Fig 6.6: table of results

6.2.4.7 Statistical study

For the I/O operations, as can be seen in figure 6.6, the degradation of performance is ranged from 61 to 97 % for the magnetic units. The card reader and the console are special cases which will be examined further. Here, it has been impossible to compute a ponderation for each type of I/O operations. But, as all the I/O operations are initated by the same privileged instruction (SDV: start device), it has been possible to quantify the number of SDV executed the same way we quantify the number of each privileged instruction executed (cfr 6.2.4.2).

Computing of the statistical data:

$$\text{Mean degradation} = \frac{1}{7} \sum_{i=1}^{7} \text{degradation}_i$$

$$= 157.6\ \%$$

$$\text{Total degradation} = \sum_{i=1}^{7} \text{degradation}_i$$

$$= 1103\ \%$$

Total number of SDV executed = 6 569

$$\text{Ratio of SDV} = \frac{\text{total number of SDV}}{\text{total number of instruc.}} = 0.0054$$

Relative degradation of I/O operation = Ratio of SDV * total degradation

$$= 6.037\ \%$$

For the I/O's on console, the degradation may seem important (697%), but the two units used for the measurements were different. For the measurement of the real system, we used the real console of the system, that is to say a fast unit connected on afast line. For the mesurements of the simulated system, we used a normal terminal, that is to say a relatively slow unit connected on a slow line (compared to the one used for the console of the system).

For the card reader, the degradation may seem small (23%), but as a card reader is a slow unit, the time taken by the VMM for the conversion of the I/O operation is small in comparison with the time taken by the physical and mechanical operation on the card reader.

6.2.4.8 Conclusion for the I/O operation measurements

A relative degradation of 6.037 % for all the I/O operation realized when simulating a BS1000 under BS2000 + SIM7000 instead of running it on a bare machine is relatively low. This quite comprehensible because it does not use virtual memory system and there are no overheads due to paging operations. Moreover, all the I/O operations measured by the self-loading are simple ones. For the I/O operations realized by a conventional operating systems, they more complex and thus may take much time to be translated by the VMM in order to be executable. Thus, the degradation for the I/O operations may be a little bit more important than 6 %.

CONCLUSION

7 Conclusion

Virtual machine systems were theoretically developed to correct some of the shortcomings of the typical third generation architectures and multi-programming operating systems. After being, for a certain number of years academic curiosities, they are now seen as cost-effective techniques for organizing computer systems to provide system flexibility and support for certain unique applications. Constant researches are made to improve I/O control mechanisms, sharing resources among virtual machines and formulation of resources allocation policies in order to provide efficient virtual machine operations.

On the standpoint of performance, it seems that complex operating systems will always run somewhat more slowly in a virtual machine system than in its real counterpart. This resulting throughput degradation must be carefully weighed against the benefits obtained through the use of virtual machine systems.

Virtual machine systems have several implications for overall system reliability. Perhaps the most important one is the extremelly high degree of isolation that a VMM provides for each virtual machine running under its control. A software failure in one virtual machine will not affect the functioning of the other independant virtual machines, even if the failure results from an error in simulated system code. Thus, the VMM can localize, control and isolate the impact of simulated systems' errors just the same way a conventional multi-programming system does it for the users' program errors. This is due to the fact that the virtual machine systems are three-level hierarchically structured instead of two levels generally found in conventional operating systems. Furthermore, by using redundant security mechanisms, a high degree of reliability is attainable.

Availability may be also used as a indicator of reliabilty when comparing two equivalent systems: OS-n and VMM/OS-1. After having characterized the overall value of the systems in terms of a utility function defined on all the possible degrees of availalbility, we haver demonstrated a number of conditions under which virtual machine systems would be superior to comparable multi-programming systems organized in the conventional manner.

Another advantage of virtual machine system is its great versattility and flexability due to the fact that it is possible to run two distinct nuclei on the same hardware at the same time. This can alleviate the new release "trauma" by permitting system generation and schooling of the new release simultaneously with production uses of the old release. It also allows the development of anew operating system while the old one is used for production schedule.

Virtual machine systems also provide the ability to work with a virtual configuration which can be quite different from the real one. As all the I/O operation issued by the simulated system are translated and converted, it is possible to map I/O from a device to another.

A last advantage of virtual machine systems is the possibility for any terminals in the configuration to become the console of the simulated system and for the users to become the operator of its system.

In conclusion, virtual machine systems have some important advantages for special applications but these ones may be severely curtails when thinking to the degradation of performance when running in a virtual environment.

APPENDIX

8 Appendix 1

Balance equations:

(1) $dP_i' = cP_i$ $(0 \leq i \leq n)$

(2) $(ai+(n-i)b+c)P_i = (n-i+1)bP_{i-1} + (i+1)aP_{i+1} + dP_i'$ $(1 \leq i \leq n-1)$

(3) $(nb + c)P_0 = aP_1 + dP_0'$

(4) $(na + c)P_n = bP_{n-1} + dP_n'$

Solving of these equations:

(1) $dP_i' = cP_i$ ===> $P_i' = \frac{c}{d} P_i$ $(0 \leq i \leq n)$

(1) and (3) ===> $P_1 = \frac{nb}{a} P_0$

(1),(2) and i=1 ===> $(a+(n-1)b)P_1 = nbP_0 + 2aP_2$

$(anb/a + n(n-1)\, b^2/a)P_0 = nbP_0 + 2aP_2$

$P_2 = \frac{1}{2a} (nb + \frac{n(n-1)b^2}{a} - nb)P_0$

$P = \frac{n(n-1)}{2} \frac{b^2}{a^2} P_0$

(1),(2) and i=2 ===> $(2a+(n-2)b)\ \frac{n(n-1)}{2}\ \frac{b^2}{a^2}\ P_o =$

$$n(n-1)b\ \frac{b}{a}\ P_o + 3aP_3$$

$$P_3 = \frac{n(n-1)(n-2)\ b^3}{6\ a^3}\ P_o$$

==========> $P_i = C_n^i \left[\frac{b}{a}\right]^i P_o \qquad (1 \leq i \leq n-1)$

(4) ===> $P_n = 1/n\ b/a\ P_{n-1} = 1/n\ b/a\ C_{n-1}^n \left[\frac{b}{a}\right]^{n-1} P_o$

$$= \left[\frac{b}{a}\right]^n P_o$$

Computing of P_o:

The normalizing condition is:

$$\sum_{i=0}^{n} P_i + \sum_{i'=0}^{n} P_{i'} = 1$$

By spliting the different sums, we have:

$$P_o + \sum_{i=1}^{n-1} P_i + P_n + P_{o'} + \sum_{i'=1}^{n-1} P_{i'} + P_{n'} = 1$$

As $P_{i'} = c/d\ P_i$ (equation (1)), we have:

$$P_o + \sum_{i=1}^{n-1} P_i + P_n + c/d\ P_o + c/d \sum_{i=1}^{n-1} P_i + c/d\ P_n = 1$$

By replacing P_i and P_n in function of P_o; we have:

$$P_o + \sum_{i=1}^{n-1} C_n^i \left[\frac{b}{a}\right]^i P_o + \left[\frac{b}{a}\right]^n P_o + \frac{c}{d} P_o + \frac{c}{d} \sum_{i=1}^{n-1} C_n^i \left[\frac{b}{a}\right]^i P_o + \frac{c}{d} \left[\frac{b}{a}\right]^n P_o = 1$$

$$P_o \left(1 + \frac{c}{d}\right) \left(1 + \sum_{i=1}^{n-1} C_n^i \left[\frac{b}{a}\right]^i + \left[\frac{b}{a}\right]^n \right) = 1$$

$$P_o \left(1 + \frac{c}{d}\right) \left(\sum_{i=0}^{n} C_n^i \left[\frac{b}{a}\right]^i \right) = 1$$

$$P_o \left(1 + \frac{c}{d}\right) \left(1 + \frac{b}{a}\right)^n = 1$$

$$P_o = \frac{d}{c+d} \left(\frac{a}{a+b}\right)^n$$

Computing of P_i and P_n:

$$P_i = C_n^i \left[\frac{b}{a}\right]^i \frac{d}{c+d} \left(\frac{a}{a+b}\right)^n$$

$$= C_n^i \frac{d}{c+d} \frac{b}{(a+b)} \frac{a}{(a+b)}$$

$$= C_n^i \frac{d}{c+d} \left[\frac{b}{a+b}\right]^i \left[\frac{b}{a+b}\right]^{n-i}$$

$$P_n = \left[\frac{b}{a}\right]^n \frac{d}{c+d} \left[\frac{a}{a+b}\right]^n$$

$$= \frac{d}{c+d} \left[\frac{b}{a+b}\right]^n$$

In conclusion, we have:

- $P_i = \frac{d}{c+d} C_n^i \left[\frac{b}{a+b}\right]^i \left[\frac{a}{a+b}\right]^{n-i}$ $\quad 0 \leq i \leq n$

- $P_{i+1} = \frac{c}{d} P_i$ $\quad 0 \leq i \leq n$

BIBLIOGRAPHY

9 Bibliography

General discussions and surveys.

[1] R.P. Goldberg, "Architecture of Virtual Machines",
Honeywell Information System Inc,
Billerica, Massachusetts, 1973.

[2] R.P. Goldberg, "Survey of Virtual Machine Research",
Honeywell Information System Inc,
Billerica, Massachusetts, 1974.

[3] R.P. Goldberg, "Virtual Machine: Semantics and Examples",
IEEE Computer Society Conference,
Boston, Massachusetts, 1971.

[4] Piperakis, "Virtual Machine: Analysis of Aspects and Applications"
Siemens Munich, Software Development Project,
vol 8,chap 40, sect 24, 1977.

[5] J.D. Bagley, "Sharing Data and Services in a Virtual Machine",
Operating System Review, vol 9 nro 5,1975.

[6] R.P. Parmelee, T.I. Peterson, C.C. Tilleman, D.J. Hattfield,
"Virtual Storage and Virtual Machines",
IBM System Journal, nro 2, 1972.

[7] G. Popeck, R. Goldberg, "Formal requirements for virtualizable
third generation architectures",
Communications of the ACM, vol 17, nro 7, july 1974

[8] K. Fuchi, H. Tanaka and T. Yuba, "A program simulator by
partial interpretation",
Second symposium on Operating System Principles,
Princeton University, Oct. 1969.

[9] R.J. Srodawa, L.A. Bates, "An efficient virtual
machine implementation"
Wayne State University
Detroit, Michigan, 48202

[10] J.P. Buzen, O. Gagliardi, "The evolution of virtual machine architecture"
Honeywell Information Systems Inc.
Billerca, Massachusetts, 1973

[11] R.A. Meyer, I.H. Seawright, "A virtual machine time sharing system"
IBM System Journal, nro 9, 1970

[12] R.J. Creasy, "The origin of the VM/370 time-sharing system"
IBM Journal of Research, vol 25, nro 5, 1981

[13] IBM, "Virtual machine facility /370: introduction"
Order nro GC20-1800-9

[14] J.P. Buzen, P.P. Chen, R.P. Goldberg, "Virtual Machine techniques for improving system reliability",
Honeywell Information system Inc,
Billerica, Massachusetts, 1973.

[15] J.J. Donovan, S.E. Madnick, "Hierarchical approach to computer integrity"
IBM System Journal, nro 2, 1975

[16] C.R. Attanasio, "Penetrating an operating system: a study of VM/370 integrity"
IBM System Journal, nro 1, 1976

[17] C.R. Attanasio, "Virtual control storage - security measure in VM/370"
IBM System Journal, nro 1, 1979

[18] P. Calloway, "Performance consideration for the use of virtual machine capabilities"
IBM Thomas J. Watson Research Center
Yorktown Heights, New York 10598

[19] C.J. Young, "Extended architecture and hypervisor performance"
Proc. ACM SIGARCH-SIGOPS
Workshop on virtual computer system
Cambridge, 1973

[20] I. Barzovsky, "Reliability theory and practice",
Prentice-Hall, 1961.

Siemens SIM7000.

[21] SIEMENS, "BS200 Introduction"
General documentation D14/4387/101

[22] D TS, "Common functional characteristics"
Siemens AG A26219-Y3-V1-3-7659
1982

[23] SWN22 Team, "SIM7000: permanent user description"
Software development project
vol 29, chap 52, sect 21, 1983

[24] SWN22 Team, "Permanent design description of the
SIM7000 product"
Software development project
vol 29, chap 61, sect 20, 1983

TABLE OF CONTENTS

TABLE OF CONTENTS

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX
INSTITUT D'INFORMATIQUE

CONCEPT OF VIRTUAL MACHINE

MODULES OF MEASUREMENTS

Mémoire présenté par

Patrick Leroy

en vue de l'obtention
du titre de
Licencié et Maître en informatique

Année académique 1983 - 1984

1 Introduction

This paper describes the special tool used to measure the degradation of performance of SIM7000 of SIEMENS. This tool is a self-loading program written in SIEMENS assembler. That is, it is loaded by the IPL procedure of the machine from a peripherical unit (a tape or a disk for instance). This program may be decomposed into two different parts: the initialisation of the hardware and the program of measurement itself. These two parts are here presented and commented.

Here, we will just present the written code in its minimal form, that is to say without any object code, litterals, flags and addresses. If the reader wants to get more information about this code, he will find complements of information in the assembler listing joined to this document.

2 Initialisation of the hardware

As the program is a self-loading program and runs without any operating system, the first operation to do is to initialize all the hardware registers. These registers are called processor utility registers. They are storage locations which are invisible to the programmer but are used by the CPU for various operations. They are generally known as pcounter, interrupt mask register, interrupt status register, segment table address register, etc...

Warning: the SIEMENS computer works with 4 different states instead of 2 for the other machines (privileged and user state). This 4 states are called P1, P2, P3 and P4.

P1: user state

P2: interruptible privileged state

P3: uninterruptible privileged state

P4: uninterruptible privileged state reserved for power failure and machine checks.

All this 4 states have their own scratchpad memory which must be initialized.

2.1.1 Code of the initialisation

```
*************************************************************************
*                                                                       *
*              LOAD SUBSYSTEM SCRATCH PAD                               *
*              --------------------------                               *
*     THIS SECTION RUNNING IN P1 & P3 STATES CONTAINS THE               *
*     INITIALISATION OF THE HARDWARE REGISTERS.                         *
*     THE SCRATCH PAD IS INITIALIZED AS FOLLOW:                         *
*                                                                       *
*        P1 PCR = P1CLEAR                                               *
*        P2 PCR = P2                                                    *
*        P3 PCR = P3CLEAR                                               *
*                                                                       *
*************************************************************************
*
*
USING  *,0
LEPVM     CCPU    EXIOCBIT,$XLCPC          LOAD CPU CONTROL REGISTER
          FCAL    IFRSAVE,$XSTIF           SAVE IFR
          FCAL    CPUID,$XSTID             SAVE CPU-IDENTIFIER
          LSP     $XP3IMR(1,0),P3LOAD0     INITIALIZE P3-IMR 1 & 2
          LSP     $P3ISR(2,0),P3LOAD1      INITIALIZE P3-ISR & P3-PCR
          LSP     $P1ISR(2,0),P1LOAD1      INITIALIZE P1-ISR & P1-PCR
          LSP     $P2ISR(2,0),P2LOAD1      INITIALIZE P2-ISR & P2-PCR
          LSP     $XP2IMR(1,0),P2LOAD0     INITIALIZE P2-IMR 1 & 2
          LSP     $XP1IMR(1,0),P1LOAD0
          PC      P1CLEAR,$P3              ENTER IN P3 STATE
*
*  IMR ISR & PCR VALUES
*
          DC      0F
P3LOAD0   DC      X'0000001300000007'      ONLY P4 INTERRUPTS ALLOWED
P3LOAD1   DC      XL4'00'
          DC      AL4(P3CLEAR)
P1LOAD0   DC      X'FFFEFE1FFFFFFFFF'      ALMOST ALL INTERRUPTS ALLOWED
P1LOAD1   DC      XL4'00'
          DC      AL4(P1CLEAR)
P2LOAD1   DC      XL4'00'
          DC      AL4(P2)
P2LOAD0   DC      X'FFFEFE1FFFFFFFFF'      ALMOST ALL INTERRUPTS ALLOWED
          DS      0D
```

```
EXIOCBIT  DC      X'40000000A0000000'
IFRSAVE   DS      D
CPUID     DS      D


* THE FOLLOWING PC STATEMENTS RUN ALTERNALY IN P1 & P3 IN
* ORDER TO RESET THE P1-ISR
*
P3CLEAR   EQU     *
          PC      P3CLEAR,$P1
P1CLEAR   EQU     *
          LSP     $P3ISR(2,0),P3ADR
          PC      P1CLEAR,$P3
          DS      0F
P3ADR     DC      XL4'00'
          DC      AL4(P3INIT)
P3INIT    PC      P3ROUTIN,$P2                    EN $P2 AT P2
```

2.2 Bootstrap procedure

After the initialition of the hardware, we must read the rest of the self-loading program on the tape. This is due to the fact that the IPL procedure read just the first block on the tape, store it in main memory at the real address 0 and give the control at this address. Here, we also find the code of the introduction of the CUU (channel and unit number) of all the unit used by the self-loading program. This introduction is realized throught the console of the system.

2.2.1 Code of the procedure

```
*****************************************************************
*            BOOTSTRAP PROCEDURE: LOAD THE 2ND BLOCK            *
*            ---------------------------------------            *
*  THIS SECTION TURNS IN P2 STATE IN ORDER TO:                  *
*      1) LOAD THE SECOND BLOCK FROM TAPE                       *
*      2) ENTER THE CUU OF THE USED UNITS                       *
*                                                               *
*****************************************************************
*
P2          LA     R2,4
            LA     R3,CARACT
            XR     R4,R4
LOOPCUU     LA     R11,CUUCCW                 CCW TO READ FROM CONSOLE
            MVC    MSGCUU+10(4),0(R3)
            BAL    R14,INOUT
            PACK   STARTIME,CUU               CONVERSION OF CUU INTO
            CVB    R10,STARTIME               BINARY
            STH    R10,DEVICEAD(R4)
            LA     R3,4(R3)
            LA     R4,2(R4)
            BCT    R2,LOOPCUU
            LA     R11,BOOTCCW                CCW TO READ THE 2ND BLOCK
            LA     R1,12                      FROM THE TAPE
            LH     R12,IPLDVAD
            SDV    0(R12)
            IDL    0
CONTBOOT    B      PC
*
* DATA OF THE MAIN ROUTINE
*
STARTIME    DS     D
ENDTIME     DS     D
CARACT      DC     C'IPL '
            DC     C'DISK'
            DC     C'TAPE'
            DC     C'CARD'
DEVICEAD    DS     0H
IPLDVAD     DS     H
DISKDVAD    DS     H
TAPEDVAD    DS     H
```

```
CARDDVAD  DS    H
CUUCCW    CCW   X'03',MSGCUU,X'40',X'23'
CCW   X'05',CUU,X'20',X'04'
BOOTCCW   CCW   X'05',X'1000',X'20',X'1000'
CUU       DS    F
MSGCUU    DC    X'15',C'? CUU OF IPL  ? (DEC. IN 4 CHAR!!)'
```

2.3 Instruction PC (Program Control)

This instruction specifies the termination of program execution in the current processor state and the initiation of an other state specified in the instruction. The current processor state is desactived and its pcounter is initialized with the address specified in the instruction and the new processor state is actived at the address specified by its pcounter.

2.3.1 Example

Let's suppose we are in P3 state and that we want to execute

```
PC      ADR1,$P2
```

That will cause:

- activation of the state P2
- save of the P3 pcounter in the scratchpad of P3 state the current state
- branch to the address contained in the P2 pcounter of the P2 scratchpad memory

2.3.2 Code of the measurement

```
*************************************************************************
*                  MEASURE OF PC INSTRUCTION                            *
*************************************************************************
*
*
PC          XR      R8,R8
            LA      R9,100
            LSP     $XP3PCR,=A(P3PC)          LOAD P3PCR WITH A(P3PC)
LOOP        STCK    STARTIME                  STORE THE TIME
            PC      RET,$P3                   PC INSTR TO BE MEASURED
RET         BAL     R14,COMPTA                   | BRANCH AT P3PCR IN
            BCT     R9,LOOP                      | P3 STATE AND LOAD
            LSP     $XP3PCR,=A(P3ROUTIN)         | P2PCR WITH A(RET)
            MVC     MSG+24(18),=C' PC  INSTRUCTIONS '
            BAL     R14,CONVERT
            B       LSP1
*
*
P3PC        STCK    ENDTIME                   STORE THE TIME
            PC      P3PC,$P2                  BRANCH AT P2PCR IN P2 STATE
                                              AND LOAD P3PCR WITH A(P3PC)
```

2.4 Instruction LSP (Load Status of Program)

This instruction loads the CPU registers from a field located in main storage. This instruction has a lot of subfunctions which allow to load almost any field of the scratchpad memory. Here, we will just measure 3 of them: the loading of one word, the loading of the general registers and the loading of all the context of one program.

2.4.1 Example

Let's suppose we want to execute

```
LSP     $XP3PCR,=A(P3ROUTIN)
```

That will cause:

- the load of the P3 pcounter in the P3 scratchpad memory with the address of P3ROUTIN.

2.4.2 code of the measurement

```
*****************************************************************
*                MEASURE OF LSP  INSTRUCTION                    *
*****************************************************************
*
*
LSP1        XR     R8,R8
            LA     R9,100
LLSP1       STCK   STARTIME                  STORE THE TIME
            LSP    $XP3PCR,=A(P3ROUTIN)      LOAD 1 WORD OF THE SCRPD
            STCK   ENDTIME                   STORE THE TIME
            BAL    R14,COMPTA
            BCT    R9,LLSP1
            MVC    MSG+24(18),=C'LSP (1 WORD)      '
            BAL    R14,CONVERT
            B      LSP16
*
*
LSP16       XR     R8,R8
            LA     R9,100
            SSP    $XP3GR,CONTEX
LLSP16      STCK   STARTIME                  STORE THE TIME
            LSP    $XP3GR,CONTEX             LOAD 16 REG OF THE SCRPD
            STCK   ENDTIME                   STORE THE TIME
            BAL    R14,COMPTA
            BCT    R9,LLSP16
            MVC    MSG+24(18),=C'LSP (16 WORDS)    '
            BAL    R14,CONVERT
            B      LSPALL
*
*
LSPALL      XR     R8,R8
            LA     R9,100
            SSP    $XP3PCTX,CONTEX
LLSPALL     STCK   STARTIME                  STORE THE TIME
            LSP    $XP3PCTX,CONTEX           LOAD ALL THE CONTEXT
            STCK   ENDTIME                   STORE THE TIME
            BAL    R14,COMPTA
            BCT    R9,LLSPALL
            MVC    MSG+24(18),=C'LSP (ALL CONTEXT) '
            BAL    R14,CONVERT
```

```
          B      SSP1
*
* DATA OF LSP MEASURE
*
          DS     0D
CONTEX    DS     48F
```

2.5 instruction SSP (Store Status of Program)

This instruction stores the CPU registers into a field located in main storage. This instruction has a lot of subfunctions which allow to store almost any field of the scratchpad memory. Here, we will just measure 3 of them: the storing of one word, the storing of the general registers and the storing of all the context of one program.

2.5.1 Example

Let's suppose we want to execute

```
SSP     $XP2PCR,SAVEP2PC
```

That will cause

- the store of the P2 pcounter of the P2 scratchpad memory into the word named SAVEP2PC

2.5.2 Code of the measurement

```
*****************************************************************
*                MEASURE OF SSP  INSTRUCTION                    *
*****************************************************************
*
          DS    OH
SSP1      XR    R8,R8
          LA    R9,100
LSSP1     STCK  STARTIME                STORE THE TIME
          SSP   $XP2PCR,CONTEX          STORE 1 WORD
          STCK  ENDTIME                 STORE THE TIME
          BAL   R14,COMPTA
          BCT   R9,LSSP1
          MVC   MSG+24(18),=C'SSP (1 WORD)      '
          BAL   R14,CONVERT
          B     SSP16
*
*
SSP16     XR    R8,R8
          LA    R9,100
LSSP16    STCK  STARTIME                STORE THE TIME
          SSP   $XP2GR,CONTEX           STORE P2 REGISTERS
          STCK  ENDTIME                 STORE THE TIME
          BAL   R14,COMPTA
          BCT   R9,LSSP16
          MVC   MSG+24(18),=C'SSP (16 WORDS)    '
          BAL   R14,CONVERT
          B     SSPALL
*
*
SSPALL    XR    R8,R8
          LA    R9,100
LSSPALL   STCK  STARTIME                STORE THE TIME
          SSP   $XP2PCTX,CONTEX         STORE ALL THE CONTEXT
          STCK  ENDTIME                 STORE THE TIME
          BAL   R14,COMPTA
          BCT   R9,LSSPALL
          MVC   MSG+24(18),=C'SSP (ALL CONTEXT) '
          BAL   R14,CONVERT
          B     LSAL
```

2.6 Instruction FCAL (Function Call)

This instruction has several special fuctions covered by the same operation code. The special function to be performed is specified in the instruction itself. We will here examine the principal functions covered by this instruction.

2.6.1 Subfunction LSAL (Load Segment Table and Address Length)

This instruction loads the segment table address register (STAR) together with the segment table length register from a word located in main storage.

2.6.1.1 Example

Let's suppose we want to execute

```
FCAL      ADRSTAR,$XLSAL
```

That will cause:

- the load of the harware regisetr STAR with the address and trhe length contained in the word ADRSTAR located in main storage
- $XLSAL is a special code which allows to specify the special subfunction of FCAL to be performed.

2.6.1.2 Code of the measurement

```
*********************************************************************
*                MEASURE OF LSAL INSTRUCTION                        *
*********************************************************************
*
DS    0H
LSAL      XR    R8,R8
          LA    R9,100
LOOPLSAL  FCAL  AHLSAL,$XSSAL          STORE THE ACTUAL STAR
          STCK  STARTIME               STORE THE TIME
          FCAL  AHLSAL,$XLSAL          LOAD THE STAR WITH AHLSAL
          STCK  ENDTIME                STORE THE TIME
          BAL   R14,COMPTA
          BCT   R9,LOOPLSAL
          MVC   MSG+24(18),=C'LSAL INSTRUCTIONS '
          BAL   R14,CONVERT
          B     SSAL
*
* DATA OF LSAL MEASURE
*
          DS    0H
AHLSAL    DC    AL2(FWLSAL)
FWLSAL    DS    F
```

2.6.2 Subfunction SSAL (Store Segment table Address and Length)

This instruction stores the segment table address register (STAR) together with the segment table length into a word located in main storage.

2.6.2.1 Example

Let's suppose we want to execute:

```
FCAL      ADRSTAR,$XSSAL
```

That will cause:

- the store of the actual address of the segment table and length into the word ADRSTAR located into the main memory.
- $XSSAL is a code which allows to specifie the special subfunction of FCAL to be performed.

2.6.2.2 Code of the measurement

```
*************************************************************************
*                  MEASURE OF SSAL INSTRUCTION                          *
*************************************************************************
*
            DS    OH
SSAL        XR    R8,R8
            LA    R9,100
LOOPSSAL    STCK  STARTIME                  STORE THE TIME
            FCAL  AHSSAL,$XSSAL             STORE SEG. TABLE ADR
            STCK  ENDTIME                   STORE THE TIME
            BAL   R14,COMPTA
            BCT   R9,LOOPSSAL
            MVC   MSG+24(18),=C'SSAL INSTRUCTIONS '
            BAL   R14,CONVERT
            B     STIF
*
* DATA OF SSAL MEASURE
*
            DS    OH
AHSSAL      DC    AL2(FWSSAL)
FWSSAL      DS    F
```

2.6.3 Subfunction STIF (Store Interrupt Flag register)

This instruction stores the interrupt flag register of the CPU executing the instruction STIF into a double word located in main storage. The content of the interrupt flag register remains unchanged.

2.6.3.1 Example

Let's suppose we want to execute:

```
FCAL     STSTIF,$XSTIF
```

That will cause:

- the store of the interrupt flag register into the double word STSTIF located in the main memory.
- $XSTIF is a code which allows to specify the special subfunction of FCAL to be performed.

2.6.3.2 Code of the measurement

```
*************************************************************************
*                MEASURE OF STIF INSTRUCTION                            *
*************************************************************************
*
STIF       XR     R8,R8
           LA     R9,100
LOOPSTIF   STCK   STARTIME                  STORE THE TIME
           FCAL   STSTIF,$XSTIF             STORE THE IFR IN STSTIF
           STCK   ENDTIME                   STORE THE TIME
           BAL    R14,COMPTA
           BCT    R9,LOOPSTIF
           MVC    MSG+24(18),=C'STIF INSTRUCTIONS '
           BAL    R14,CONVERT
           B      TSR
*
* DATA OF STIF MEASURE
*
STSTIF     DS     D
```

2.6.4 Subfunction TSR (Test and Set Real)

This instruction tests and sets a byte located in main storage. The address specified for the byte must be the real one. The byte is read and bit 0 of this byte is used to set the code condition. Then X'FF' is stored into the byte. The code condition is set as follows:

condition code:

0 - bit 0 of the byte was 0
1 - bit 1 of the byte was 1
2 - not used
3 - not used

2.6.4.1 Example

Let's suppose we want to execute:

```
FCAL      BYTETSR,$XTSR
```

That will cause:

- the test of the byte BYTETSR located in the main memory, the test is done following the rules exposed just before.
- $XTSR is a code which allows to specifie the special subfunction of FCAL to be performed.

2.6.4.2 Code of the measurement

```
*************************************************************************
*                MEASURE OF TSR  INSTRUCTION                            *
*************************************************************************
*
*
TSR         XR     R8,R8
            LA     R9,100
LOOPTSR     STCK   STARTIME                  STORE THE TIME
            FCAL   BYTETSR,$XTSR             SET AND TEST BYTETSR
            STCK   ENDTIME                   STORE THE TIME
            BAL    R14,COMPTA
            BCT    R9,LOOPTSR
            MVC    MSG+24(18),=C'TSR  INSTRUCTIONS '
            BAL    R14,CONVERT
            B      STID
*
* DATA OF TSR MEASURE
*
BYTETSR     DC     B'00000000'
```

2.6.5 Subfunction STID (Store CPU Identification)

This instruction stores the CPU identification of the CPU executing the STID instruction into a double word located in main storage.

2.6.5.1 Example

Let's suppose we want to execute:

```
FCAL        STSTID,$XSTID
```

That will cause:

- the store of the CPU identifier into the double word STSTID located into the main memory.
- $XSTID is a code which allows to specify the special subfunction of FCAL to be performed.

2.6.5.2 Code of the measurement

```
*********************************************************************
*                MEASURE OF STID INSTRUCTION                        *
*********************************************************************
*
STID       XR     R8,R8
           LA     R9,100
LOOPSTID   STCK   STARTIME                  STORE THE TIME
           FCAL   STSTID,$XSTID             STORE THE CPU IDENT
           STCK   ENDTIME                   STORE THE TIME
           BAL    R14,COMPTA
           BCT    R9,LOOPSTID
           MVC    MSG+24(18),=C'STID INSTRUCTIONS '
           BAL    R14,CONVERT
           B      STNU
*
* DATA OF STID MEASURE
*
STSTID     DS     D
```

2.6.6 Subfunction STNU (Store CPU Number)

This instruction stores the content of the CPU number into one byte located in the main storage. The CPU number is a one byte field defined during installation of a multi-processor configuration to uniquely identify each CPU in the configuration. The number will be binary encoded.

2.6.6.1 Example

Let's suppose we want to execute:

```
FCAL        STSTNU,$XSTNU
```

That will cause:

- the store of the CPU number into the byte STSTNU located in the main memory.
- $XSTNU is a code which allows to specify the special subfunction of FCAL to be performed.

2.6.6.2 Code of the measurement

```
*********************************************************************
*                  MEASURE OF STNU INSTRUCTION                      *
*********************************************************************
*
          DS    0H
STNU      XR    R8,R8
          LA    R9,100
LOOPSTNU  STCK  STARTIME                  STORE THE TIME
          FCAL  STSTNU,$XSTNU             STORE CPU NUMBER
          STCK  ENDTIME                   STORE THE TIME
          BAL   R14,COMPTA
          BCT   R9,LOOPSTNU
          MVC   MSG+24(18),=C'STNU INSTRUCTIONS '
          BAL   R14,CONVERT
          B     LDWR
*
* DATA OF STNU MEASURE
*
STSTNU    DS    CL1
```

2.6.7 Subfunction LDWR (Load Word Real)

This instruction loads a word into a general register specified in the instruction from the main storage area designated by a real word oriented address contained in an other general register also specified in the instruction.

2.6.7.1 Example

Let's suppose we want to execute:

```
FCAL        16*R11+R10,$XLDWR
```

That will cause:

- the load into register 11 of the word located at the real address specified by the register 10.
- $XLDWR is a code which allows to specify the special subfunction of FCAL to be performed.

2.6.7.2 Code of the measurement

```
*************************************************************************
*                MEASURE OF LDWR INSTRUCTION                            *
*************************************************************************
*
*
LDWR       XR     R8,R8
           LA     R9,100
           LA     R10,WORDLDWR
LOOPLDWR   STCK   STARTIME                  STORE THE TIME
           FCAL   16*R11+R10,$XLDWR         LOAD WORD IN R11 FROM R10
           STCK   ENDTIME                   STORE THE TIME
           BAL    R14,COMPTA
           BCT    R9,LOOPLDWR
           MVC    MSG+24(18),=C'LDWR INSTRUCTIONS '
           BAL    R14,CONVERT
           B      LDHR
*
* DATA OF LDWR MEASURE
*
WORDLDWR   DS     F
```

2.6.8 Subfunction LDHR (Load Halfword Real)

This instruction loads a halfword into bits 16-31 of a general register specifed in the instruction from a real halfword oriented address contained in an other general register also specified in the instruction. Bits 0-15 of the first general register are set to zeros.

2.6.8.1 Example

Let's suppose we want to execute:

```
FCAL        16*R11+R10,$XLDHR
```

That will cause:

- the load into the register 11 of the halfword located at the real address specifed by the register 10.
- $XLDHR is a code which allows to specify the special subfunction of FCAL to be performed.

2.6.8.2 Code of the measurement

```
*************************************************************************
*                MEASURE OF LDHR INSTRUCTION                            *
*************************************************************************
*
*
LDHR       XR     R8,R8
           LA     R9,100
           LA     R10,HALFLDHR
LOOPLDHR   STCK   STARTIME                  STORE THE TIME
           FCAL   16*R11+R10,$XLDHR         LOAD HALF IN R10 FROM R11
           STCK   ENDTIME                   STORE THE TIME
           BAL    R14,COMPTA
           BCT    R9,LOOPLDHR
           MVC    MSG+24(18),=C'LDHR INSTRUCTIONS '
           BAL    R14,CONVERT
           B      STWR
*
* DATA OF LDHR
*
HALFLDHR   DS    H
```

2.6.9 Subfunction STWR (Store Word Real)

This instruction stores a word from the general register specified in the instruction into the main storage area designated by the real word oriented address contains in an other general register also specified in the instruction. The contents of the general registers are unchanged.

2.6.9.1 Example

Let's suppose we want to execute:

```
FCAL      16*R11+R10,$XSTWR
```

That will cause:

- the store of the register 11 into a word located at the real address specified by the register 10.
- $XSTWR is a code which allows to specify the special subfunction of FCAL to be performed.

2.6.9.2 Code of the measurement

```
*************************************************************************
*                  MEASURE OF STWR INSTRUCTION                          *
*************************************************************************
*
*
STWR        XR     R8,R8
            LA     R9,100
            LA     R10,WORDSTWR
LOOPSTWR    STCK   STARTIME                    STORE THE TIME
            FCAL   16*R11+R10,$XSTWR           STORE WORD FROM R10 TO R11
            STCK   ENDTIME                     STORE THE TIME
            BAL    R14,COMPTA
            BCT    R9,LOOPSTWR
            MVC    MSG+24(18),=C'STWR INSTRUCTIONS '
            BAL    R14,CONVERT
            B      STHR
*
* DATA OF STWR MEASURE
*
WORDSTWR    DS     F
```

2.6.10 Subfunction STHR (Store Halfword Real)

This instruction stores bits 16-31 of a general register specified in the instruction into the main storage area designated by the real halfword oriented address contained in an other general register also specified in the instruction. The contents of the general registers remain unchanged.

2.6.10.1 Example

Let's suppose we want to execute:

```
FCAL        16*R11+R10,$XSTHR
```

That will cause:

- the store of the register 11 into a halfword located in main memory at the real address specified by hte register 10.
- $XSTHR is a code which allows to specify the special subfunction of FCAL to be performed.

2.6.10.2 Code of the measurement

```
*************************************************************************
*                MEASURE OF STHR INSTRUCTION                            *
*************************************************************************
*
*
STHR       XR     R8,R8
           LA     R9,100
           LA     R10,HALFSTHR
LOOPSTHR   STCK   STARTIME                    STORE THE TIME
           FCAL   16*R11+R10,$XSTHR           STORE HALF FROM R10 TO R11
           STCK   ENDTIME                     STORE THE TIME
           BAL    R14,COMPTA
           BCT    R9,LOOPSTHR
           MVC    MSG+24(18),=C'STHR INSTRUCTIONS '
           BAL    R14,CONVERT
           B      STIO
*
* DATA OF STHR MEASURE
*
HALFSTHR   DS     H
```

2.6.11 Subfunction STIO (Store I/O Status)

This instruction store the four words of the I/o status information in main memory at an address which must be word aligned. The four words contain succesively the CAR (Channel Address Register), the CCR2 (Channel Command Register 2), the CCR1 (Channel Command Register 1) and the DSR (Device Status Register).

2.6.11.1 Example

Let's suppose we want to execute:

```
FCAL        STSTIO,$XSTIO
```

That will cause:

- the store of the I/O status into 4 words located in the main memory at the address STSTIO.
- $XSTIO is code which allows to specify the special subfunction of FCAL to be performed.

2.6.11.2 Code of the measurement

```
*************************************************************************
*                 MEASURE OF STIO INSTRUCTION                           *
*************************************************************************
*
*
STIO       XR     R8,R8
           LA     R9,100
LOOPSTIO   STCK   STARTIME                    STORE THE TIME
           FCAL   FWSTIO,$XSTIO               STORE THE I/O STATUS
           STCK   ENDTIME                     STORE THE TIME
           BAL    R14,COMPTA
           BCT    R9,LOOPSTIO
           MVC    MSG+24(18),=C'STIO INSTRUCTIONS '
           BAL    R14,CONVERT
           B      TDV
*
* DATA OF STIO MEASURE
*
FWSTIO     DS     4F
```

2.7 Instruction TDV (Test Device)

The data path specified in the instruction is checked wether it could be initiated by a SDV or not. If there is any condition in the I/O components which could prevent the success of command initiating, the instruction is terminated with a condition code different from 0 and reponse information is given in register 12. If command initiating would succeed, the instruction terminates with a condition code equal to 0.

2.7.1 Example

Let's suppose we want to execute:

```
TDV       0(R12)
```

That will cause:

- the analyse of the data path specified by the register 12
- the postionement of the condition code following the result of the check.

2.7.2 Code of the measurement

```
*************************************************************************
*                MEASURE OF TDV  INSTRUCTION                            *
*************************************************************************
*
*
TDV         XR     R8,R8
            LA     R9,100
LOOPTDV     LH     R12,CONSDVAD           CONSDVAD = CUU OF THE CONSOLE
            STCK   STARTIME               STORE THE TIME
            TDV    0(R12)                 TEST CONSOLE CHANNEL
            STCK   ENDTIME                STORE THE TIME
            BAL    R14,COMPTA
            BCT    R9,LOOPTDV
            MVC    MSG+24(18),=C'TDV  INSTRUCTIONS '
            BAL    R14,CONVERT
            B      SDV
```

2.8 Input/output instruction

For Siemens computer, all the I/O operations are done with the same instruction, SDV (Start Device). This instruction uses two operands, the CAW (Channel Address Word) and the CCW (Channel Command Word). The CAW must contain the channel number (C) and the unit number (UU) in the form CUU. The CCW is a double word containing the I/O command to be executed in the form of CMD, ADR, CHAIN, LENGTH.

CMD: I/O command code

ADR: address of data

CHAIN: code of chaining several CCW

LENGTH: length of the data

For the SDV to be executed correctly, the register 12 must contain the CAW and the register 11 must contain the address of the CCW chain to be executed.

2.8.1 Input/output on console

The CAW of the console is equal to 000
The CCW is equal to

X'03',TESTMSG,X'20',X'11'

with: X'03' = write command
TESTMSG = address of the message to write
X'20' = no chain command
X'11' = length of the message in hexadecimal

2.8.1.1 Code of the measurement

```
*************************************************************************
*                MEASURE OF SDV  (WRITE ON CONSOLE)                     *
*************************************************************************
*
*
SDV         XR     R8,R8
            LA     R9,100
            LA     R11,TESTCCW              CCW TO WRITE ON CONSOLE
LOOPSDV     LA     R1,1
            LH     R12,CONSDVAD             CONSDVAD = CUU OF CONSOLE
            STCK   STARTIME                 STORE THE TIME
            SDV    0(R12)
            IDL    0
CONTIN      XR     R1,R1
            MVC    ENDTIME(8),SAVETIME
            BAL    R14,COMPTA
            BCT    R9,LOOPSDV
            MVC    MSG+24(18),=C'SDV ON CONSOLE    '
            BAL    R14,CONVERT
            B      SDVTAPEW
*
* DATA OF SDV MEASURE
*
TESTCCW     CCW    X'03',TESTMSG,X'20',X'11'
TESTMSG     DC     X'15',C'MEASURE OF 1 SDV'
```

2.8.2 Input/output on tape

2.8.2.1 Write on tape

The CAW of tape is equal to TAPEDVAD which is asked at the begining of the program.

The CCW is equal to

X'03',X'1800',X'20',X'800'

with X'03' = command code for write

X'1800' = real address of I/O buffer

X'20' = no chain command

X'800' = length of I/O buffer in hexadecimal (2 Kb)

2.8.2.1.1 Code of the measurement

```
*****************************************************************
*                MEASURE OF SDV  (WRITE ON TAPE)                *
*****************************************************************
*
*
SDVTAPEW  XR    R8,R8
          LA    R9,100
          LA    R1,2
          LA    R11,TINITCCW      CWW OF INITIALISATION ON TAPE
          LH    R12,TAPEDVAD      TAPEDVAD = CUU OF TAPE
          SDV   0(R12)
          BC    7,FIN
          IDL   0
CONTAPMA  LA    R11,TAPECCWW      CCW TO WRITE ON TAPE
          LA    R1,3
          LH    R12,=H'6160'      6160 = A(I/O BUFFER + 16)
          ST    R9,0(R12)         STORE RECORD NUM INTO I/O BUFFER
          LH    R12,TAPEDVAD
          STCK  STARTIME          STORE THE TIME
          SDV   0(R12)
          BC    7,FIN
          IDL   0
CONTAPEW  MVC   ENDTIME(8),SAVETIME
          BAL   R14,COMPTA
          BCT   R9,CONTAPMA
          MVC   MSG+24(18),=C'WRITE 2 KB ON TAPE'
          BAL   R14,CONVERT
          B     SDVTAPER
*
* DATA OF SDV (WRITE ON TAPE)
*
TAPECCWR  CCW   X'03',X'1800',X'20',X'800'
TINITCCW  CCW   X'07',BOT,X'20',X'01'
          DS    0D
```

2.8.2.2 Read on tape

The CCW is equal to

X'05',X'1800',X'20',X'800'

with X'05' = command code for read

X'1800' = address of the I/O buffer

X'20' = no command chaining

X'800' = length of I/O buffer

2.8.2.2.1 Code of the measurement

```
*****************************************************************
*                MEASURE OF SDV  (READ ON TAPE)                 *
*****************************************************************
*
*
SDVTAPER  XR     R8,R8
          LA     R9,100
          LA     R1,5
          LA     R11,TINICCWR          CCW TO REWIND THE TAPE
          LH     R12,TAPEDVAD
          SDV    0(R12)
          BC     7,FIN
          IDL    0
CONTTPM   LA     R11,TAPECCWR          CCW TO READ ON TAPE
          LA     R1,6
          LH     R12,TAPEDVAD
          STCK   STARTIME              STORE THE TIME
          SDV    0(R12)
          BC     7,FIN
          IDL    0
CONTAPER  MVC    ENDTIME(8),SAVETIME
          BAL    R14,COMPTA
          BCT    R9,CONTTPM
          MVC    MSG+24(18),=C'READ 2 KB ON TAPE '
          BAL    R14,CONVERT
          B      SDVDISKW
*
* DATA OF SDV (READ ON TAPE)
*
TAPECCWR  CCW    X'05',X'1800',X'20',X'800'
TINICCWR  CCW    X'07',BOT,X'20',X'01'
```

2.8.3 Input/output on disk

2.8.3.1 Write on disk

The CAW of the disk is equal to DISKDVAD which is asked at the console at the begining of the program.

The CCW is equal to

```
X'27',MMCCHHR,X'40',X'06'
X'53',MMCCHHR+2,X'40',X'05'
X'09',SEARCHW,X'40',X'00'
X'A3',X'1800',X'20',X'800'
```

with: X'27' = command code for seek on disk

X'53' = " " for search on disk

X'09' = " " for tic on disk

X'A3' = " " for write on disk

MMCCHHR = 7 bytes to define the address on disk
MM for magasin
CC for cylinders
HH for heads
R for records

2.8.3.1.1 Code of the measurement

```
*****************************************************************
*                MEASURE OF SDV  (WRITE ON DISK)                *
*****************************************************************
*
*
SDVDISKW  XR     R8,R8
          XR     R2,R2
          XR     R9,R9
          MVC    MMCCHHR(7),=X'00000001000001'
DISKWRT   LA     R9,1(R9)
          CH     R9,=H'101'
          BNL    ENDWDISK
          LA     R1,7
          LA     R11,DISKCCWW       CCW FOR WRITE ON DISK
          LH     R12,=H'6160'       6160 = A(I/O BUFFER + 16)
          ST     R9,0(R12)          STORE RECORD NUM INTO I/O BUFFER
          LH     R12,DISKDVAD       DISKDVAD = CUU OF DISK
          STCK   STARTIME           STORE THE TIME
          SDV    0(R12)
          IDL    0
CONTDISW  MVC    ENDTIME(8),SAVETIME
          BAL    R14,COMPTA
          IC     R2,MMCCHHR+6
          AH     R2,=H'1'
          STC    R2,MMCCHHR+6       INCREMENTATION OF THE
          CH     R2,=H'8'           RECORD NUMBER
          BH     INCRHH
          B      DISKWRT
*
INCRHH    IC     R2,MMCCHHR+5
          AH     R2,=H'1'
          STC    R2,MMCCHHR+5
          MVC    MMCCHHR+6(1),=X'01'    INCREMENTATION OF THE
          CH     R2,=H'8'               HEAD NUMBER
          BH     INCRCC
          B      DISKWRT
*
```

```
INCRCC    IC     R2,MMCCHHR+3
          AH     R2,=H'1'
          STC    R2,MMCCHHR+3                INCREMENTATION OF THE
          MVC    MMCCHHR+4(3),=X'000001'     CYLINDER NUMBER
          B      DISKWRT
*
ENDWDISK  MVC    MSG+24(18),=C'WRITE 2 KB ON DISK'
          BAL    R14,CONVERT
          B      SDVDISKR


*
* DATA OF SDV (WRITE ON DISK)
*
DISKCCWW  CCW    X'27',MMCCHHR,X'40',X'06'
SEARCHW   CCW    X'53',MMCCHHR+2,X'40',X'05'
          CCW    X'09',SEARCHW,X'40',X'00'
          CCW    X'A3',X'1800',X'20',X'800'
MMCCHHR   DS     CL7
```

2.8.3.2 Read on disk

The CAW and THE CCW are just the same as those used for write on disk but the command code A3 is replaced by the command code A5 to read on disk.

2.8.3.2.1 Code of the measurement

```
*****************************************************************
*               MEASURE OF SDV  (READ ON DISK)                  *
*****************************************************************
*
*
SDVDISKR  XR     R2,R2
          XR     R8,R8
          XR     R9,R9
          MVC    MMCCHHR(7),=X'00000001000001'
DISKRD    LA     R9,1(R9)
          CH     R9,=H'101'
          BNL    ENDRDISK
          LA     R1,8
          LA     R11,DISKCCWR            CCW TO READ ON DISK
          LH     R12,DISKDVAD
          STCK   STARTIME
          SDV    0(R12)
          IDL    0
CONTDISR  MVC    ENDTIME(8),SAVETIME
          BAL    R14,COMPTA
          IC     R2,MMCCHHR+6            INCREMENTATION OF THE
          AH     R2,=H'1'                RECORD NUMBER
          STC    R2,MMCCHHR+6
          CH     R2,=H'8'
          BH     INCRHHR
          B      DISKRD
*
```

```
INCRHHR   IC     R2,MMCCHHR+5
          AH     R2,=H'1'
          STC    R2,MMCCHHR+5            INCREMENTATION OF THE
          MVC    MMCCHHR+6(1),=X'01'     HEAD NUMBER
          CH     R2,=H'8'
          BH     INCRCCR
          B      DISKRD
*
INCRCCR   IC     R2,MMCCHHR+3
          AH     R2,=H'1'                  INCREMENTATION OF THE
          MVC    MMCCHHR+4(3),=X'000001'   CYLINDER NUMBER
          B      DISKRD
*
ENDRDISK  MVC    MSG+24(18),=C'READ 2 KB ON DISK '
          BAL    R14,CONVERT
          B      SDVCARD


*
* DATA OF SDV (READ ON DISK)
*
DISKCCWR  CCW    X'27',MMCCHHR,X'40',X'06'
SEARCHR   CCW    X'53',MMCCHHR+2,X'40',X'05'
          CCW    X'09',SEARCHR,X'40',X'00'
          CCW    X'A5',X'1800',X'20',X'800'
```

2.8.4 Input on card reader

The CAW of the card reader is equal to CARDDVAD which is asked at the console at the begining of the program.

The CCW is equal to

X'05',CARDBUFF,X'20',X'50'

with X'05' = command code for read a card

CARDBUFF = address of the buffer

X'20' = no command chain

X'50' = length of a card in hexadecimal

2.8.4.1 Code of the measurement

```
*************************************************************************
*                MEASURE OF SDV  (READ A CARD)                          *
*************************************************************************
*
*
SDVCARD    XR     R8,R8
           LA     R9,100
CARDRD     LA     R1,9
           LA     R11,CARDCCW              CCW TO READ A CARD
           LH     R12,CARDDVAD             CARDDVAD = CUU OF READER
           STCK   STARTIME
           SDV    0(R12)
           IDL    0
CONTCARD   MVC    ENDTIME(8),SAVETIME
           BAL    R14,COMPTA
           BCT    R9,CARDRD
           MVC    MSG+24(18),=C'READ 1 CARD 80 CAR'
           BAL    R14,CONVERT
           B      PAGING
*
* DATA OF SDV (READ A CARD)
*
CARDCCW    CCW    X'05',CARDBUFF,X'20',X'50'
CARDBUFF   DS     CL80
```

2.9 Operation of paging

2.9.1 Explanation

The concept of virtual memory is implemented on the BS2000 system by use of a technique called "paging". Paging is the transfert of 4096-bytes blocks (pages) of programs between the auxiliary storage device and main memory as they are needed for processing.

Here, to measure the time taken by an operation of paging, this one has programmed in P3 state and realized following the general principals of paging operation and we force the operation to occur by executing an operation of move in a page not present in main memory.

2.9.2 Code of the measurement

```
*************************************************************************
*                  MEASURE OF PAGING                                    *
*************************************************************************
*
*
PAGING     XR     R8,R8
           LA     R9,50
           LSP    $XP3PCR,=A(PAGINIT)
           LSP    $P3R14,=F'20'
           PC     CONTPAG,$P3
*
PAGINIT    L      R5,=A(X'40000')
           FCAL   ADRSTAR,$LBTP          SET VIRTUAL ADRESS ON
           PC     P3ROUTIN,$P2
*
CONTPAG    LA     R10,2
PAGING02   L      R13,=A(X'2000')        START WITH PAGE 5
           LA     R9,50
PAGING50   STCK   STARTIME
           MVC    0(16,R13),TEST         INSTRUC. WHICH CAUSE PAGING
           STCK   ENDTIME
           BAL    R14,COMPTA
           A      R13,=A(X'800')         INCREMENTATION OF 1 PAGE
           BCT    R9,PAGING50            LOOP OF 50 PAGES
           BCT    R10,PAGING02           LOOP OF 2 * (50 PAGES)
           MVC    MSG+24(18),=C'PAGING OPERATIONS '
           BAL    R14,CONVERT
           B      FIN
*
* DATA OF PAGING MEASURE
*
           DS     0F
ADRSTAR    DC     AL2(STAR)
           DS     0F
STAR       DC     XL2'1000'
           DC     XL2'1000'
TEST       DC     C'TEST  TEST  TEST'
```

2.10 Conversion routine

This routine is used to convertr the time used by 100 instructions in a printable form and to print it on the console of the system.

2.10.1 Code of the routine

```
*********************************************************************
*      THIS P2 ROUTINE IS ACTIVED FOR CONVERSION OF THE             *
*      EXECUTION TIME OF 100 MEASURED INSTRUCTIONS AND DISPLAY      *
*      THE RESULT ON CONSOLE.                                       *
*        - PRECONDITION: R8 = EXECUTION TIME TO CONVERT             *
*********************************************************************
*
*
CONVERT   CVD   R8,ENDTIME              CONVERT THE TIME IN DEC.
          MVC   TIMEOUT(14),MASK        CONVERT THE TIME IN
          ED    TIMEOUT(14),ENDTIME+2   PRINTABLE FORM
          LA    R11,MSGCCW              CCW TO WRITE ON CONSOLE
          BAL   R14,INOUT
          BR    R14
*
* DATA OF THE CONVERT ROUTINE
*
MSG       DC    X'15',C' EXECUTION TIME OF 100      INSTRUCTIONS :'
TIMEOUT   DS    XL14
          DC    C' SEC.'
MASK      DC    X'4020202021206B2020204B202020'
MSGCCW    CCW   X'03',MSG,X'20',X'3E'
SAVEADR   DS    F
```

2.11 Comptabilisation routine

This routine is used to compute the time taken by one instruction The two double words STARTIME and ENDTIME contain the value of the clock at the begining and at the end of the instruction to measure. These two values are substracted and added to register 8 containing the cumuml.

2.11.1 Code of the routine

```
*************************************************************************
*         THIS P2 ROUTINE CONVERT THE TIME OF ONE MEASURED              *
*         INSTRUCTION AND ADD TO R8                                     *
*                                                                       *
*           - PRECONDITION: CALLED BY BAL        R14,COMPTA             *
*                                                                       *
*************************************************************************
*
*
COMPTA     LM     R4,R5,STARTIME          CONVERT THE TIME
           LM     R6,R7,ENDTIME           OF ONE MEASURED
           SRDL   R4,12                   INSTRUCTION INTO BINARY
           SRDL   R6,12                   AND ADD IT TO R8 WHICH
           SR     R7,R5                   CONTAINS THE CUMUL OF THE
           AR     R8,R7                   EXECUTION TIME
           BR     14                      RETURN
```

2.12 Interupt analyse routine

This routine is actived by any occurence of an interruption in the system. It analyses the cause of the interruption, treat it completely and reactivate at the correct address the P2 code to be executed. Here, only two causes of interruption are analysed and treated, all the other are rejected and caused an abnormal end to occur. These two causes are the end of I/O and the paging interruption.

2.12.1 Code of the routine

```
*************************************************************************
*         THIS P3 ROUTINE IS ONLY ACTIVED BY INTERRUPTIONS              *
*                                                                       *
*    - INTERRUPT WEIGHT = FROM 24 TO 3C   ==> END OF I/O                *
*                         4C              ==> PAGING                    *
*                                                                       *
*************************************************************************
*
*
P3ROUTIN  STCK   SAVETIME
          C      R15,=A(X'4C')            PAGING INTERRUPT ?
          BE     PAGQUEUE                 YES,BRANCH TO ANALYZE OF PAGI
          MVC    INTMSG+17(14),=C' IW < 24 !!    '
          C      R15,=A(X'24')
          BL     ERRINT                                  END I/O INTERUPT ?
          MVC    INTMSG+17(14),=C' IW > 3C !!    '
          C      R15,=A(X'3C')                           NO, BRANCH TO ERRO
          BH     ERRINT                                  ROUTINE
          FCAL   SAVESTIO,$XSTIO                         SAVE I/O STATUS
          MVC    INTMSG+17(14),=C'I/O STA  >< 48'
          TM     SAVESTIO+15,X'48'                       I/O NORMAL END?
          BNO    ERRINT                                  NO,BRANCH TO ERROR
          MVC    INTMSG+17(14),=C' TAKE A DUMP!!'
          SSP    $P2R1,REG1
          L      R7,REG1                    R1 = SWITHCH TO RETURN
          MH     R7,=H'4'                   AT THE CORRECT PLACE IN P2
          L      R11,TABBR(R7)
          BR     R11
TESTSDV0  LSP    $XP2PCR,=A(CONT)
          B      RETURN
TESTSDV1  LSP    $XP2PCR,=A(CONTIN)
          B      RETURN
TESTSDV2  LSP    $XP2PCR,=A(CONTAPMA)
          B      RETURN
TESTSDV3  LSP    $XP2PCR,=A(CONTAPEW)
          B      RETURN
TESTSDV4  LSP    $XP2PCR,=A(SDVTAPER)          RETURN IN P2 BY
          B      RETURN                        MEANS OF A TABLE
TESTSDV5  LSP    $XP2PCR,=A(CONTTPM)           OF ADDRESSES
          B      RETURN
```

```
TESTSDV6   LSP    $XP2PCR,=A(CONTAPER)
           B      RETURN
TESTSDV7   LSP    $XP2PCR,=A(CONTDISW)
           B      RETURN
TESTSDV8   LSP    $XP2PCR,=A(CONTDISR)
           B      RETURN
TESTSDV9   LSP    $XP2PCR,=A(CONTCARD)
           B      RETURN
WRITVIRT   B      SAUTWRT
READVIRT   B      SAUTRD
READBOOT   LSP    $XP2PCR,=A(CONTBOOT)
           B      RETURN
SDVTM      LSP    $XP2PCR,=A(CONTAPMA)
           B      RETURN
SDVTMRD    LSP    $XP2PCR,=A(CONTTPM)
           B      RETURN
*
RETURN     PC     P3ROUTIN,$P2              RETURN TO $P2 AT P2 PCR
```

```
PAGQUEUE  SSP   $XP2ERCR,SAVEERCR     COMPUTING OF THE SEGMENT
          SSP   $XP2PCR,SAVEP2PC      AND PAGE NUMBER MISSIN
          ST    R14,SAVECPT
          XR    R12,R12
          L     R13,SAVEERCR
          SLL   R13,19
          SLDL  R12,8                 R12 = SEGMENT NUMBER
          SRL   R13,27                R13 = PAGE NUMBER
          STM   R12,R13,SEGNUM
          LH    R7,PRESENT            SAVE THE PAGE?
          CH    R7,=H'0'
          BE    SAUTWRT               NO
          LM    R12,R13,OLDSEG
          BAL   R14,PTENTREE          REWRITE OF THE PAGE
          MVC   0(2,R7),=H'0'
          LSP   $XP2PCR,=A(PAGEWRT)
          PC    P3ROUTIN,$P2
*
PAGEWRT   LA    R11,DISKCCWW
          LH    R12,DISKDVAD
          LA    R1,10                 REWRITE OF THE PAGE
          SDV   0(R12)
          IDL   0
*
SAUTWRT   MVC   MMCCHHR(2),=X'0000'
          LM    R12,R13,SEGNUM
          LR    R7,R12
          MH    R7,=H'32'
          AR    R7,R13                    COMPUTING OF THE MMCCHHR
          MVC   MMCCHHR+2(2),=X'0001'
          CH    R7,=H'72'
          BNH   SAUTCC2
          SH    R7,=H'72'
          MVC   MMCCHHR+2(2),=X'0002'
SAUTCC2   MVC   MMCCHHR+4(3),=X'000000'
          XR    R14,R14
          LR    R15,R7
          D     R14,=F'8'
          STC   R15,MMCCHHR+5
          AH    R14,=H'1'
          STC   R14,MMCCHHR+6
          LSP   $XP2PCR,=A(PAGERD)
          MVC   PRESENT(2),=H'1'
          L     R14,SAVECPT
```

```
            BCT    R14,SAUTINCR
            LA     R14,20
            LH     R15,SEARCHW+18
            A      R15,=A(X'800')
            STH    R15,SEARCHW+18
            STH    R15,SEARCHR+18
            LH     R15,PAGENUM
            AH     R15,=H'1'
            STH    R15,PAGENUM
SAUTINCR    PC     P3ROUTIN,$P2
*
PAGERD      LA     R11,DISKCCWR
            LH     R12,DISKDVAD
            LA     R1,11                    READ OF THE PAGE
            SDV    0(R12)
            IDL    0
*
SAUTRD      LM     R12,R13,SEGNUM
            STM    R12,R13,OLDSEG
            BAL    R14,PTENTREE
            MVC    0(2,R7),PAGENUM
            L      R11,SAVEP2PC             COMPUTING OF THE P2 PCOUNTER
            LR     R12,R11                  TO RESTART THE INSTRUCTION
            SRL    R12,30                   WHICH CAUSED AN PAGING
            SLL    R12,1
            SLR    R11,R12
            ST     R11,SAVEP2PC
            LSP    $XP2PCR,SAVEP2PC
            PC     P3ROUTIN,$P2             RESTART OF THE INSTRUCTION
*
ERRINT      LSP    $XP2PCR,=A(ERREUR)
            PC     P3ROUTIN,$P2
*
ERREUR      LA     R11,INTCCW               CCW TO WRITE AN ERROR MESSAGE
            BAL    R14,INOUT                ON CONSOLE
            B      FIN
*
PTENTREE    MH     R12,=H'4'
            AH     R12,=H'4096'
            L      R7,0(R12)                COMPUTING OF THE ENTRY IN
            SLL    R7,8                     SEGMENT AND PAGE TABLES
            SRL    R7,8
            MH     R13,=H'2'
            AR     R7,R13
```

```
          BR     R14


*
* DATA OF P3ROUTINE
*
SAVETIME  DS     D
SAVEERCR  DS     F
SAVESTIO  DS     4F
SAVEP2PC  DS     F
SEGNUM    DS     2F
OLDSEG    DS     2F
TABBR     DC     A(TESTSDV0)
          DC     A(TESTSDV1)
          DC     A(TESTSDV2)
          DC     A(TESTSDV3)
          DC     A(TESTSDV4)
          DC     A(TESTSDV5)
          DC     A(TESTSDV6)
          DC     A(TESTSDV7)          TABLE OF ADDRESS OF RETURN
          DC     A(TESTSDV8)
          DC     A(TESTSDV9)
          DC     A(WRITVIRT)
          DC     A(READVIRT)
          DC     A(READBOOT)
          DC     A(SDVTM)
          DC     A(SDVTMRD)
REG1      DS     F
SAVECPT   DS     F
PRESENT   DC     H'0'
PAGENUM   DC     H'8195'              8195 = X'2003'
INTCCW    CCW    X'03',INTMSG,X'20',X'20'
INTMSG    DC     X'15',C' INTERRUPT ERROR, TAKE A DUMP!!'
```

2.13 Input/output routine

This routine is used to read or to write any data to or from the console of the system. Before calling it, the register 11 must contain the address of the CCW to be executed.

2.13.1 Code of the routine

```
*************************************************************************
*           THIS P2 ROUTINE IS ACTIVED WHEN AN I/O ON CONSO             *
*           IS DESIRED.                                                 *
*              - PRECONDITION: R11 CONTAINS THE CCW CHAIN ADRESS        *
*                              CALLED BY BAL       R14,INOUT            *
*************************************************************************
*
*
INOUT     LH     R12,CONSDVAD          CONSDVAD = CUU OF CONSOLE
          XR     R1,R1
          SDV    0(R12)
          BC     7,FIN
          IDL    0
CONT      BR     R14
*
* DATA OF INOUT ROUTINE
*
CONSDVAD  DC     H'0'
```

2.14 termination routine

This routine is used to stop the execution of the program. in fact, the program is pended in P3 state which is not interruptable, so that the program is waiting indefinitely.

2.14.1 Code of the routine

```
*************************************************************************
*           THIS P2ROUTINE IS ACTIVED WHEREVER IT IS DESIRE             *
*           TO STOP THE EXECUTION OF THE PROGRAM.(NORMAL OR             *
*           ABNORMAL END)                                               *
*************************************************************************
*
*
FIN         LSP    $P3PCR,=A(P3STOP)        TASK PENDED IN P3 WHICH
            PC     PC,$P3                   UNINTERRUPTABLE
P3STOP      IDL    0
```

2.15 I/O buffer and translation table

The I/O buffer has the address X'1800' (in hexadecimal) and has a length of 2048 bytes. The translation table used by the paging operation has 32 entry points containing each the address of the page table corresponding to this entry.

```
*
* I/O BUFFER OF 2K AND TRANSLATION TABLE
*
ORG   LEPVM+X'1000'                    256 WORDS ALIGNEMENT!!
SEGTAB     DC     X'C0'
           DC     AL3(PAGTAB1)
           DC     X'C0'
           DC     AL3(PAGTAB2)
           DC     X'C0'
           DC     AL3(PAGTAB3)
           DS     29F
PAGTAB1    DC     X'2000'
           DC     X'2001'
           DC     X'2002'
           DC     X'2003'
           DC     28H'0'
PAGTAB2    DC     32H'0'
PAGTAB3    DC     32H'0'
           ORG    LEPVM+X'1800'
IOBUFFER   DC     C'I/O BUFFER OF 2K'
           DC     240C' '
           DS     7CL256
```

TABLE OF CONTENTS