Abstract:
The role of the "Supervisor-state, Problem-state" dichotomy in machine architecture is examined. These two states are implemented in machines to facilitate the protection of vital information contained in hardware registers or memory which, if improperly changed, would affect the correct operation of the system. The same kind of protection can be achieved with address mapping techniques resembling those found in existing hardware. Thus the supervisor-state and the privileged operations defined by it can be eliminated, and all programs at all levels of the system can be run on machines with an identical hardware order code. In particular, a copy of an operating system may be run as a job under itself. An example is presented of the design of a machine with a base-limit form of address relocation and no supervisor state. It is conjectured that similar techniques could be applied to machines with more sophisticated structures.

IS SUPERVISOR-STATE NECESSARY?

By

Hugh C. Lauer
and C.R. Snow

TECHNICAL REPORT SERIES

Series editor: Dr. B. Shaw

Number 30.
December, 1971

© 1971 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
LAUER, Hugh C.

Is Supervisor-state necessary? [By]
Hugh C. Lauer and C. R. Snow.
Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1971.
(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series No. 30)

12"

Added entries
SNOW, Charles Richard
UNIVERSITY OF NEWCASTLE UPON TYNE.

Suggested classmarks (primary classmark underlined)
Library of Congress:
   Dewey (17th): 621.381952
   U.D.C.: 681.322.02

Suggested keywords
MACHINE ARCHITECTURE
OPERATING SYSTEMS
PROTECTION STRUCTURES
SUPERVISOR STATE
VIRTUAL MEMORY

Abstract
The role of the "Supervisor-state, Problem-state" dichotomy in machine architecture is examined. These two states are implemented in machines to facilitate the protection of vital information contained in hardware registers or memory which, if improperly changed, would affect the correct operation of the system. The same kind of protection can be achieved with address mapping techniques resembling those found in existing hardware. Thus the supervisor-state and the privileged operations defined by it can be eliminated, and all programs at all levels of the system can be run on machines with an identical hardware order code. In particular, a copy of an operating system may be run as a job under itself. An example is presented of the design of a machine with a base-limit form of address relocation and no supervisor state. It is conjectured that similar techniques could be applied to machines with more sophisticated structures.

About the Author
Mr. Lauer has been a Lecturer in Newcastle University Computing Laboratory since January 1971, before which he was a Ph.D. candidate in Computer Science at Carnegie-Mellon University, Pittsburgh, Penn.

Mr. Snow has been a Lecturer in the same Department since August 1970 prior to which he was a Research Associate studying "Computer Representation of Linear Graphs".
With but a few exceptions, all large computing systems of the last
decade have incorporated two principal states of operation in their central
processors. Various named supervisor state and problem state, master mode
and guard or slave mode, and privileged mode and unprivileged mode, these
states are designed to protect programs from each other and guarantee that
the operating system retains ultimate control over the system resources.
The effect is that certain processor registers and operations are available
only to programs operating in the privileged mode. Non-privileged programs
(i.e., those operating in the non-privileged state) are restricted in their
use of resources and in communication with other programs, and they must ask
an all-knowing operating system to perform essential services for them.

This dichotomy imposes some programming burdens. For example,
privileged and non-privileged programs run on machines of slightly different
definition, and so cannot be made to obey exactly the same programming
conventions. Privileged programs also cannot be tested in non-privileged
mode under the control of an operating system except when elaborate
interpretive schemes are provided (e.g. the CP67 system [10]). Another
difficulty is that the two states rarely provide discriminating protection
among the various parts of a large control program, all of which must
operate in privileged mode. Instead administrative controls are used
during the programming process to provide the protection. As a result of
these difficulties, the privileged programs in operating systems are
relatively static and unchanging, even when they should not be.

In recent years, various authors have developed methods to rationalise
the design of very large, very complex operating systems, particularly by
building hierarchically structured systems (see Dijkstra [6] and Zurcher and
Randell [12]). The presence of a privileged mode imposes an essential
difference between the hardware level program and those of other levels.
In addition, it defines boundaries between some levels which might be
unnatural. Thus in the systems described by Dijkstra, Brinch Hansen [1]
and Lampson [9], there is no explicit privileged mode. Protection is
achieved within the addressing structure by preventing programs from having
have presented processor designs based on this kind of protection.

These considerations suggest that the privileged mode may be unnecessary
and may, in fact, get in the way of good system design. Fabry makes this
point in his thesis, and his design eliminates the privileged mode for all
processor functions except the residual one of protecting the address mapping
mechanism. In this paper, we conjecture that the privileged mode can be
eliminated entirely. To support the conjecture we will outline the design
of a simple processor in which the various programs are protected only by the
addressing function. To provide motivation, we will first examine the
function of privileged modes and present a point of view to characterise
hierarchical systems.

The Function of the Privileged Mode

From the definitions of typical central processors ([3],[4],[8]), we
find the operations protected by a privileged mode fall into four categories:
1) Commands to input/output devices and other processors.
2) Commands to enable or disable interrupts.
3) Commands to change the address mapping function or tables.
4) Commands to change the state of the processor itself.

At least two machines, the Modular ONE [3] and the PDP-11 [5], have already eliminated the operations of the first category by making device control registers addressable in the same way as memory locations. Control over the devices is achieved by reading and writing these locations, and protecting input/output devices is reduced to the problem of protecting any location in addressable memory. Commands of the second category can be treated exactly like those of the first. I.e., the control registers for the interrupt system can be made addressable and protected by the memory protection scheme.

Commands of the third and fourth categories are more complicated. For motivation, let us consider a type of hierarchically organized system in which the processors for each level are identical. The CP67 system [10] is an example: the "virtual machines" defined by (the most recent version of) the operating system are logically identical to the IBM 360/67 hardware [8] and are thus capable of running copies of the same system. This can be repeated recursively to any depth.

Figure 1 illustrates the function of address mapping hardware in forming the hierarchy. Portions of the core memory have been allocated to one or more programs of the next level. These are called "virtual" memories. Associated with each is an address mapping function in the form of a data structure which describes how to translate virtual memory addresses into real core memory addresses. In practice, this function may consist of values representing the base and limit of a virtual memory [4], segment and page tables [8], or a more complex structure [2]. The general function of the address mapping mechanism is to 1) provide for each virtual memory a naming scheme independent of all other contexts, and 2) insure that programs in virtual memories do not go outside of them.

Now suppose that within a virtual memory, we wish to create another one - a "virtual-virtual" memory - in the same way that the system created ours. To do this, we specify an address mapping function which maps virtual-virtual addresses into virtual addresses. The system then composes this mapping with that which maps virtual addresses to real core addresses. Figure 2 illustrates with the virtual memories and maps indicated by level numbers.

There are three distinct contexts in which a given address can be interpreted: level 0, level 1, or level 2. The processor must always be told which one is in force. Typically the operations necessary to do this are in terms of absolute addresses and therefore privileged. Thus, programs in levels 1 and 2 cannot execute them directly but must call upon interpretive routines declared in level 0 to do the necessary work. Now if we wish to test a copy of the operating system in a level 1 virtual memory, we have two levels of interpretation. Level 2 programs attempt to execute privileged operations which are interpreted by level 1. In these routines, the latter attempts to execute privileged operations which are then interpreted by level 0. In an n level system, there would be n levels of interpretation.
In practice, the interrupt structure of most machines makes it more difficult than this. An attempt to execute a privileged operation at level \( n \) results in a program-fault trap directly to level 0. This recognizes that it must simulate a program-fault trap in level 1, and it executes privileged operations to do so. Level 1 contains an identical copy of the operating system and so must simulate a program-fault trap in level 2. Since this involves privileged operations, it traps back to level 0 where these are interpreted. Similarly, at level \( n-1 \), the original program fault is interpreted using privileged operations which require interpretation at previous levels. Thus the context switching required to service the \( n \)th level resembles an \( n \)-level Tower of Hanoi problem.

We can characterize this structure by an analogy with Algol. Each virtual machine can be regarded as a block with scope rules reversed so that a program cannot access variables of a containing block but can access variables of inner blocks. The hardware processor itself is the outermost block and its internal registers are the variables of that block. Scope rules for procedure declarations and calls are the same as in Algol. Each operation in the processor instruction set can be regarded as a call upon a procedure at an appropriate level which performs the operation. The non-privileged operations are declared within the processor, but different privileged operation procedures are declared at each level. In effect, a program at level \( n \) calls a procedure declared at level \( n-1 \) to interpret each privileged operation, and these procedures operate upon variables declared at the same level. At the hardware level, these procedures check a program's privilege before proceeding. Indeed, the reason for the privileged mode is to prevent programs from indiscriminately changing critical hardware variables such as address mapping registers.

**A Machine Design**

With this point of view, we can design a processor with an address mapping mechanism which does not have to be protected from programs at any level. Thus, commands of the third category need not be privileged. Once this is accomplished, commands of the fourth category can also be made non-privileged since there will no longer be any privileged state to protect. The principal characteristic of our design will be that no programmable register or memory location will contain an absolute address. Instead, all addresses will be relative to the context that generates them. A program at any level will be able to declare a virtual machine inside of it and to pass control to that machine without losing ultimate control or requiring the assistance of an operating system.

We consider the central processor as a collection of procedures and register variables which are used to interpret programs. It is described in an appropriate language which we choose to be an informal combination of Algol and English. For simplicity, we will assume that programmable registers such as accumulators, index registers, instruction counters, etc., are part of the addressable virtual memory of each process (as, for example, the accumulators in the EDP-10 [7]) and are loaded or stored in the appropriate place whenever the notations

\[
\text{load status} \quad \text{and} \quad \text{store status}
\]
appear in our description. Our description will concentrate on the address mapping function because the rest of the design is entirely conventional.

The processor will have two internal registers, called BASE and LIMIT to identify the current virtual memory in a conventional way, as illustrated by Figure 3. They will not be addressable by any programs, not even the operating system, but instead will be manipulated by the procedures described below. BASE and LIMIT contain real core addresses with BASE < LIMIT. A virtual address \( a \) is converted to a real core address simply by adding it to BASE. This is done by the following procedure in the processor:

```plaintext
hardware procedure absolute address (a);
    absolute address:= if BASE + a < LIMIT then
                        BASE + a else program fault;
```

The hardware procedure "program fault" will be described later.

Suppose a program at level \( b \) declares a virtual machine within its own space, i.e., at level \( n + 1 \). It allocates an area between locations \( b \) and \( l \), sets the initial state in the appropriate place, then passes control by executing a hardware operation TRANSFER TO VM. This instruction will set BASE and LIMIT to point to the new virtual machine, but first it leaves a trail to permit recovery at a later time. The two words immediately preceding the new machine, labelled \( \beta \) and \( \lambda \) in Figure 4, are used for this purpose. The TRANSFER TO VM function is defined by

```plaintext
hardware procedure TRANSFER TO VM (b,l);
    if 0 < b < l < LIMIT-BASE then
        begin save status;
            \( \beta := b \); \( \lambda := LIMIT-BASE-l \);
            BASE := BASE + b;
            LIMIT := BASE + l;
        load status end
        else program fault;
```

Note that the backward links stored in locations \( \beta \) and \( \lambda \) cannot be altered by the new virtual machine because they are not in its memory space. Similarly, they are safe from the calling program because they are set by the hardware after that program has given up control.

When the program in the new virtual machine either fails or calls upon the containing environment for service, control must return to that environment. The hardware routine RETURN FROM VM uses the backward links to accomplish this.

```plaintext
hardware procedure RETURN FROM VM (code);
    begin save status; -
            BASE := BASE - \( \beta \);
            LIMIT := LIMIT + \( \lambda \);
        load status and code end;
```

6.
Locations $\beta$ and $\lambda$ are now those immediately proceeding the one named by BASE. The parameter "code" is used to transmit the reason for returning control. Thus, the hardware procedure "program fault" is defined by

RETURN FROM VM (program fault code)

and a supervisor call instruction is defined by

RETURN FROM VM (supervisor call code).

Because of the structure of the backward links, neither the TRANSFER TO VM instruction nor the RETURN FROM VM instruction need be privileged. Thus any program at any level can declare and pass control to successive levels in the hierarchy while still retaining ultimate control over those levels. No supervisor intervention is required. The mechanism applies to every level of the system including the zeroth (hardware) level, where BASE has the value zero and LIMIT has a value equal to the size of memory. For this level, locations $\beta$ and $\lambda$ are "core locations" $-1$ and $-2$ and contain the values zero. Thus the zeroth level program returns control to itself in the event of a program fault.

The essential characteristic of the design is that there are no absolute addresses available to any program. The only ones are contained in BASE and LIMIT, but these can be manipulated only by the hardware routines we have defined. The backward links stored immediately before each virtual memory are not absolute themselves, although they permit the reconstruction of absolute addresses. However, no program can access any links which are still meaningful, so that BASE and LIMIT are protected from uncontrolled change.

Interrupts from input/output devices, timers, or events external to the system must be treated differently from program faults. The latter are principally directed to the immediate supervisor (i.e. the immediately containing environment) of the program generating them, but the former can be directed to an unrelated program in an unrelated environment. Thus it is often necessary to communicate the occurrence of the interrupt to a program which is not currently running. This can be done with the help of the following routine:

```
hardware procedure interrupt (code);

begin
  save status;
  BASE := 0
  LIMIT := largest core address;
  load status and code;

end;
```

This routine, in effect, throws away all of the backward links and returns control to the hardware level environment. The program at that level must examine the code and call the appropriate software routine.

To re-establish the hierarchy, control must be passed along the interrupted chain, one level at a time. Each level must recognise that it has received control as a result of a spurious interrupt. This is possible if we adopt the convention that a zero value for an interrupt-trap code means
no action is required. Thus when a program gives control to a virtual machine in its own space, it should do so with the following (software) routine:

\[
\text{while interrupt-trap code} = 0 \text{ do}
\]
\[
\text{TRANSFER TO VM (b, l);}
\]

Clearly the efficiency of a many-levelled system suffers with this design because the effort of servicing an interrupt is proportional to the depth of the hierarchy. On the other hand, communication between programs in adjacent levels is direct and handled by unprivileged hardware operations. Thus, the privileged mode can be eliminated without sacrificing any degree of protection.

**Generalisations**

The machine just presented is simple but has the properties to support our conjecture. It is more general than most existing hardware in the sense that it naturally encourages a hierarchical structure without imposing artificial boundaries or an omnipotent supervisor. But it lacks some of the important features of modern systems, particularly segmented virtual memories and a suitable parameter passing mechanism (Fabry [7]).

Segmenting a memory complicates the absolute address function and makes it difficult to find a place to store the backward links. For protection, we cannot allow programs to form segment tables with absolute addresses any more than we could allow them to load BASE and LIMIT indiscriminately. Thus segment tables must either be held in a separate memory or be formed dynamically by composing entries from tables of each level of the hierarchy. Furthermore, there is no convenient place to store a backward link to the previous level in the hierarchy because any segment within one virtual memory can be declared in another, perhaps with several different names. Thus there are no locations analogous to \( \beta \) and \( \lambda \) in the segmented scheme.

The problem of passing the names of procedures from one level to another is closely related to the problem of implementing name parameters in Algol. We have shown a way for a program to call a procedure in an adjacent level (using either \text{TRANSFER TO VM} or \text{RETURN FROM VM}), but we have no mechanism for skipping levels except by explicitly passing through each of the intervening levels. A solution to this problem would also be useful in the servicing of interrupts, as it is in the B6700 system [2].

**Conclusions**

We have presented a single machine design in which no privileged state is necessary. As a corollary to the design, no program ever has access to or need for absolute memory addresses, not even the operating system. Whether or not this scheme can be extended to useful machine designs is still an open question.

**Acknowledgements**

The authors would like to thank Professor B. Randell, Professor W.C. Lynch and Dr. A.J. Mascall for their stimulating discussions and contributions on this subject.
References


8. IBM Corporation, System 360 Model 67 Functional Characteristics, Poughkeepsie, N.Y.


