Segmentation and virtual address topology—an essay in virtual research

By J. G. Laski*

This paper exhibits the hardware logic of a two-dimensional addressing scheme. This scheme has more elaborate facilities built into the hardware than the paging logic or segmenting logic of any machine I have seen proposed elsewhere. I do not put this logic forward as a proposal for yet another machine design; my purpose rather is to describe facilities that it would be possible to embed in a machine. They thus realise various possible virtual space topologies that could be provided for the designer of the operating system. Which one he chooses depends on what will most economically give the facilities he needs. Thus when describing hardware, I suggest some uses for it and point out conceptual difficulties that it does not resolve. It is not my purpose to exhibit the logic of the kinds of operating systems that such hardware serves and which, itself, imposes what hardware is required. Fragmentary aspects will be found in the papers named in the bibliography to which the reader is recommended. Further, the necessary design process is to decide on the operating environment in which the users are to find themselves and then provide it by proper choice of software and hardware.

(First received May 1967, and in revised form November 1967)

Paging and segmentation

Both paging and segmentation are realised through some form of indirect access over which the user has no explicit control. It is essential to appreciate, however, that, although they are implemented by fundamentally similar mechanism, conceptually, paging in no way resembles segmentation. Distinguishing the two concepts has not been helped by the way in which segmentation has been implemented in some hardware designs; from the literature describing them it appears that the two concepts were certainly confused by the hardware designers and technical writers and apparently by the software designers.

The purpose of paging is to relieve the programmer of the need to manage physical store. To the programmer a paged store is indistinguishable from an unpaged store of the same address size. Apart from a different performance, in the cost/speed compromise, the programs he can write and the way they operate are in no way affected. (He can cheat, of course; knowing page size, he can organise his program to minimise page turning.)

The purpose of segmentation is to provide the programmer with an addressing space that is not homogenous but corresponds to the logical structure of his process. He has a number of entirely separate segments, with respect to each of which he may have an entirely different capability of access, each of which he may be sharing with an entirely different group of parallel processes, each of which may grow and contract independently according to his need, and each of which he must treat according to its logical type. Thus if a programmer ignores the segmentation structure of the machine, he wastes it. Segmentation affects the feasibility of programming techniques such as sharing data-objects (including code) between and within parallel on-going processes, inter-object access protection between and within parallel processes, dynamic acquisition and release of user space, dynamic connecting and snapping of inter-segment linkages, etc. Some of these facilities can be provided by software and user discipline, but it is fearfully expensive by comparison with having the right segmentation hardware. Unfortunately no one yet knows (since user experience is negligible, and simulations of doubtful validity) what is the right segmentation hardware.

The concept of virtual space

In a sense, virtual space is with the programmer as soon as he writes in a language other than absolute binary, for he can then write code which will be interpreted to fetch words from locations with names different from those he uses. However, the concept is not very useful when the bind time of this address translation occurs before the actual execution of the code (fetches from store), e.g. at compile time or at load time.

When, however, with base and limit registers or segment pointers, this interpretation occurs at run time, a new and useful possibility arises. Rather than a Von Neumann machine where the bits in any addressable location are interpretable as any type of data, dependent on the context in which it is fetched, the type of the data can be associated with the location, as named in the code from which the fetch is issued. Consequently these addresses may be structured more elaborately than the 2^n contiguous locations with identical properties of the Von Neumann machine, and it is useful to speak of the *virtual address space* and its topology within which the program operates.

* *Computer Science Dept., Upson Hall, Cornell University, Ithaca, N.Y.*

Given a single base and limit register, address space is *m* contiguous identical locations (where *m* can be chosen by the programmer) and fetch of a word outside the program area is illegal. This allows several programs of arbitrary length to coexist safely in core.

With two base and limit registers, there can be two types of address—two segments—one, say, to contain code and therefore properly allowing only instruction fetch or the operand of jump instructions (I—fetch), the other to contain data and therefore properly allowing only operand fetch or write (0—fetch). This $2 \times \omega$ topology (two segments, each of indefinitely many words) is the least complex that allows store to be shared between several processes; for the I—segments of two processes may be interpreted to the same physical store locations, while the O—segments address distinct areas of store (or conversely).

The conceptual importance of virtual space comes from more elaborate use of the two possibilities introduced above.

1. Subdividing user-space into distinct areas with distinct properties.
2. Using the same physical space as distinct user-space by one or many users.

There are two further economic advantages which come not from segmentation but from paging which was an earlier form of virtual space developed by the Atlas team.

3. Relocation of virtual store in physical store as it is paged in and out without requiring addressing change in the object code.
4. Scattered and partial loading of code according to current use (one level store).

It is vital to keep clear in one's mind these two kinds of uses of indirect store access methods. The conceptual possibilities and interest for this paper come from (1) and (2). (3) and (4) are of engineering importance in computer architecture; they may or may not be desirable in possible environments of a non-segmented machine. It is possible to have a segmented machine without paging, though I doubt very much whether there are any segmented environments in which paged architecture is not worthwhile.

Conversely, given a paged machine one can write code for it as though it were segmented, as indeed one can even on a machine with direct addressing. A great deal of confusion seems to arise from having observed such practice on, say Atlas, and not then following through the distinction.

It is far from clear, to my understanding at least, what topology of address space can best be made use of by operating system and can provide worthwhile facilities for the object programmer.

Without segmentation hardware to map bit-patterns that specify addresses in the virtual space topology dynamically to bit-patterns that access physical store, I do not understand how any operating-system can effectively permit the opportunities which come from sharing user-provided data objects (including code) between parallel processes and independent users, or can give store protection within and between processes by the type of access required, or allow controlled dynamic expansion and contraction of space used by the programs. These concepts are both deep and subtle. As they begin to be understood they will affect fundamentally our feeling for the use we can make of computing facilities and thus the design of our programming languages. In particular I am certain that the notion of a community data-base and a computer grid will remain hot-air until these concepts have been fully explored. I am sure that as we do so other concepts equally important will emerge from the fog.

I am now going to present the implementation tools for an $\omega \times \omega$ or so-called two-dimensional addressing space. I am hanging on many more bells and whistles than any system I have seen implemented or proposed elsewhere in order that the conceptual implications of the various possibilities can be brought out.

It is possible to devise address spaces with more elaborate topologies. In particular an address space that is tree-structured has been proposed elsewhere; it has a number of conceptual attractions, that ω by ω systems lack. There are difficulties, as I see it, in handling shared data-objects so as to ensure their integrity when several processes are messing around with them, and in specifying inter-segment references. I can just about see solutions to these problems for the $\omega \times \omega$ case; I only wish I could for a tree-structured virtual space of objects.

An important area that I am also going to exclude from discussion is that of the efficient development of physical address from virtual address.

If addressed data is not in primary store, I assume a mechanism to suspend the process that wants it in real time and use its processor on some other process. There will be a page-management scheme which will have a crafty and rapid way of choosing which virtual page to move to secondary store; address preparation and instruction execution will be overlapped by ingenious and reliable hardware; associative slave stores will be scattered through the processor wherever they will do the most good. We can all argue—and will all argue—about how best to do it, but the problems are technological, not intrinsic.

Pages

A page is a contiguously addressed region of physical addresses either in core or in secondary storage. A page table consists of a list of page descriptions (PDR) for contiguous virtual addresses, within a segment. A PDR contains three fields:

PPS:	Addresses of page in primary store or 'not in primary store' code.
PSS:	Address of page in secondary store.
PUB:	Page use bits. Information left behind in the course of store accessing as data for the store

management routines of the supervisor. The following may be useful, but what is actually provided is a technological problem:

EB:	Does the page exist?
LB:	Lock bit forbidding removal of the page.
WB:	Whether the page has been written to.
UC:	Use Count: how many times the page was used.

Segments

A segment is a region of virtual addresses contiguously addressed by any process that has access to it. It is the unit of information in which sharing between processes takes place. A segment may have properties that limit the access of all processes connected to it. These are maintained by *segment control bits* (SCB). What the type of data is—how the bits are interpreted in the segment, as seen by all processes connected to it—is maintained by segment type bits (STB). Limitations on access and type for each process that may be connected to it are maintained by the operating system, and thus for every connected process there may be its individual *access control bits* (ACB) and *access type bits* (ATB) on the access path to maintain such limitations and perhaps others voluntarily accepted.

A segment connected to one or more active processes will have a segment descriptor locked in primary storage or available in a paged list whose page-table is locked in primary store. The table in which it is found will either be a process segment table, in which case it is said to be owned by that process, or may be a system table in which case it is said to be a public segment. There is some case for making all segments public. However, access entails an additional indirection cycle. Therefore, segments local to a process are economically held in the segment table private to the process. A further use of this argument suggests that segments mostly accessed from some process should be held in that processes's segment-table. Also, I feel unhappy about a single public segment-table which would grow uncontrollable. It may be that here is where tree-structure addressing (thinking of the process segment-table itself as in this tree) is important in order to understand the notion that some segments are 'in the neighbourhood' of a given segment and thus more easily accessed.

Note that, for a segment whose segment-description is in the segment-table of that process (without indirection) ATB $\equiv$ STB, ACB $\equiv$ SCB. Otherwise rules for compounding the ATB and STB information and the ACB and SCB information are required.

Direct segment-descriptors must also hold the addressing limit for the segment, SLM and the segment-page-length, SPL. These may be changed by the segment supervisor and are not accessible to user programs. A user program may request a change in SLM and, by providing data, may advise the segment supervisor on how to manage core. Finally, if the segment is unpaged, control bits similar to PUB must be provided. The SCB of course, includes a count of how many processes are using the segment, etc. to help in store control for the segment page table.

Indirect segment-descriptors contain ACB and ATB for the access route to the segment from the process in whose segment-table they appear. They may point to direct segment-descriptors or to further indirect segment descriptors. More important to the logic of the operating system is the way that the address bits of the indirection are construed. Leaving aside the possibility of using index registers as well as the direct bits of the various fields, the following possibilities apply:

(1) One field construed as physical address in primary store.
(2) One field construed as offset in the same table as that in which the indirect segment description appears.
(3) Two fields, one of which gives the base of the table, the other giving the offset in the table.

The interesting problem, if many segment tables are in use, is how this first field determines the table base. If it is absolute physical address, effectively the facility is no more than that given by (1). If all segments are defined in a single public table, the first field of (3) is implicit. If processes, i.e. segment tables, are numbered by having their segment table base addresses in a system table, *the process segment*, the first field can access the required segment-table base by its process number. However, I feel there should be some more tree-like way of pointing to a segment-table in the lineage of the table in which the indirect segment-descriptor appears, and perhaps having an additional field to access a segment-table not on the direct lineage. This is an area where further research is badly needed.

ACB and SCB, ATB and STB

Precisely what facilities are provided in hardware and what use is made of them in the operating system is one of the key design decisions in a segmentation scheme. The features I describe here are possibilities whose economy or necessity will have to be established in any proposed architecture to achieve specific operating capacilities.

Access-control presents clearer issues than type-determination. It will be noticed that I have not proposed two segment-descriptors pointing to the same segment, as have been implemented on some machines. This is to ensure that no access route can skip around the SCB limitations. However, it may be that the operating system may, temporarily, give private access to a particular process if all other processes are to be blocked. The following fields, then, can usefully appear in ACB and SCB; they are compounded by or-ing along the access route.

NW:	Any process attempting a write command and encountering this bit is suspended.
NR:	Any process attempting an operand fetch (except in a jump command) and encountering this bit is suspended.

NI: Any process attempting an instruction fetch, or operand fetch for a jump command, is suspended.

EO: Any I-fetch suspends the process unless, either the preceding I-fetch for this process was from the same segment, or this fetch is to a standard (first?) word of the segment.

SA: If this field matches a field describing a supervisor-authority-status for a process, the restrictions above are ignored.

Access control for individual processes can alternatively be managed by loading process-held information into the processor which must match segment-held information for access to be permitted; I suspect this method to be less flexible but it may involve less hardware.

The use best made of STB and ATB to determine the interpretation of instruction codes and the applicability of instruction codes is less clear. It seems to me desirable to be able to vary the field-width of addressing and operand fetch. Whether a segment should be required to contain data of a single type, e.g. all 8-bit characters, or all 144-bit unnormalised floating point numbers with 22-bit exponent, or a particular record structure as specified by some description found elsewhere . . . is far from clear to me. The rules for compounding STB and ATB are again, still unclear.

Address interpretation

It remains to discuss how the addresses in object code are construed by the address-development logic to determine from where in store bit-patterns will be fetched. Firstly, it must be clearly realised that it must be possible to access outside the segments addressed directly or indirectly by segment-descriptors, but it must only be possible to do so by supervisor service. Hence addresses outside the segments in use by a process may be represented in a format quite different from running addresses. Thus loading, or to use a better term, connecting-segments to a running process, can be hedged in with all kinds of caution to be sure that rules of privacy, etc. are not being violated. These addresses can be specified by character strings for table look-up through a data-structure of dictionaries or a tree of tables or other imaginative complexities.

The essential requirement is that, however slow this first connection procedure may be (to allow the needed elaboration of reference among the retained objects that form the common database for all users of the system) subsequent in-line access through the segment-descriptor so introduced into the segment-table must be fast and straightforward for the process needing the connection.

The connection process described here supposes that intersegment references use two separate addresses, in separate virtual space. The first is a permanent database which is interpreted by software and then replaced with a local address (i.e. one that is valid for the duration of this process only and within this process only) that is hardware interpreted with speed and accuracy. I am unhappy about this distinction, particularly since in the on-line future many processes will be permanently active rather than transient as are those processes of today's batch environment which presently incite our imagination. Virtual addressing space here is an informing principle of what I believe to be a central research area on the hardware/software borderline.

However, we are principally concerned with interpreting the local address that is set as a result of connecting the global data-objects.

This involves three stages:

(1)	Given address	$\rightarrow$ own segment-descriptor
(2*a*)	Indirect segment-descriptor	$\rightarrow$ own segment-descriptor
(2*b*)	Indirect segment-descriptor	$\rightarrow$ another segment-descriptor
(3)	Direct segment-descriptor	$\rightarrow$ (page table of) physical segment.

Actual accesses are of the form:

$$1 < 2a \mid 2b >{}^{*}_{0} 3.$$

i.e. first we interpret the bits we are given as a segment descriptor, then segment-descriptor indirection can take place a non-negative number of times (usually 0), and finally we reach the actual data bits. Since the possible indirection is the phase where what is required is least clear, I leave it till last. Let me again emphasise that I am concerned here with what it might be possible and useful to provide to implement some specific desired operating environment, rather than what it is desirable and necessary to provide for relatively straightforward environments. Associative and slave stores and overlapping of address preparation with instruction executions are technical features in hardware implementation of such a scheme that determine what it is economic to include.

Given address $\rightarrow$ own segment-descriptor

A processor knows which process it is treating by knowing the base of the segment-table of that process. This process-base register may also determine the way in which virtual addresses are chopped up into fields if this is to vary from process to process. This possibility could allow the coexistence of processes, one of few large segments, the other of many small segments within the same total word length for addresses. (The use of modifiers (possibly for indirection) may make this unnecessary.)

$$\mathrm{ST}, \begin{pmatrix}\mathrm{SN} \\ \mathrm{SN, SM}\end{pmatrix}, \ \mathrm{LT}, \begin{pmatrix}\mathrm{LN} \\ \mathrm{LN, LM} \\ \mathrm{LN, LMA, LMB}\end{pmatrix}$$

A given address has four fields. The interpretation of the second depends on the value of the first, of the fourth on the third.

ST: Segment Tag: $= 0 \rightarrow$ No segment modifier.
$= 1 \rightarrow$ Segment modifier present.

SN: Segment Number:	Offset of segment descriptor in process segment-table. This may be full width if ST = 0 or part width and modified by SM.
SM: Segment Modifier Number.	
LT: Line Tag:	Addressing structure of line part of address e.g. whether direct or indirect, indirection and modifier sequence. How many modifiers etc.
LN: Line Number.	
LM: Line Modifiers:	Used in accord with LT given information.

Direct segment descriptor → physical address

D, SCB, STB, SUB, SPS, SSS, SLM.

D: = 0 → Direct Segment Descriptor.

SCB: Segment Control Bits.

STB: Segment Type Bits:	These can determine the field width of line addressing in the given address and the page size of the segment (including whether paged) and whether in primary store.
SUB: Segment Use Bits:	Information for accounting and the supervisor.
SPS: Segment Primary Store:	Base address of (page table of) segment in primary store.
SSS: Segment on Secondary Store:	Base address of segment on secondary store.

Indirect segment descriptor → segment descriptor

The address space for this indirection is the least clearly understood. The system I describe will cope with a fairly general scheme in which indirection can be either through a segment descriptor in the same segment table (i.e. in the same process) or in another table whose base is pointed to by a special system table—the process table—whose base is known to the system and whose page table is welded to core. Certain processes may be dummy in the sense that their only purpose is to provide a segment table whose contents are used indirectly by other processes without ever having a process applied to them. If all accesses are indirectly through a pseudo-segment table of this kind, we have a two-level form of the all-public segment scheme discussed earlier. The elaboration to tree-level, and access to a segment table of this process, is outside the scope of the facilities presented here. Again, what can be provided, and how it can be used seems to be a fruitful and interesting area of research.

$$\text{D, L, ACB, ATB, } \begin{pmatrix}\text{SON} \\ \text{SON, STN}\end{pmatrix}\text{, SNM}$$

D: Direct Bit: = 1 → Indirect segment descriptor.

L: Local Bit: = 0 → Access from this segment table (STN = self).
= 1 → Implies access from another segment table.

ACB: Access Control Bits.

ATB: Access Type Bits.

SON: Segment Offset Number:	Offset in segment-table.
STN: Segment Table Number:	Offset of pointer to segment table in system table of segment table (process table).
SNM: Segment Number Modifier:	Modifies SON.

It will be clearly seen that further elaboration of modification and indirection in interpreting indirect segment descriptors would be possible. I suspect that it would not be very useful.

I have given above a range of hardware features that it would be possible to supply. To describe how they might be used, and therefore to determine which features should appear requires, of course, writing a supervisor that uses them and then simulating the logic of its behaviour. This is outside the scope of my purpose here, but should be done, at least in imagination, by the reader to discover precisely what facilities in what operating environments impose what hardware organisation to make them economic or even feasible to provide for the users.

Acknowledgements

The S.R.C. (Atlas Computer Laboratory) supported the research on which this paper is based. I must also acknowledge with especial gratitude the many publications of the MULTICS group whose depth and clarity have given me so much to agree with and so much to disagree with.

Bibliography

1961

HOSIER, W. A. Pitfalls and safeguards in real-time digital systems with emphasis on programming, *IRE Trans.*, EM-8 (June 1961), pp. 99–115.

HOLT, A. W. Program organisation and record keeping for dynamic storage allocation, *Comm. ACM*, Vol. 4 (Oct. 1961), pp. 422–431.

KILBURN, T., and PAYNE, R. B. The Atlas supervisor, *Proc. AFIPS*, 1961 *Eastern Joint Comput. Conf.*, Dec., 1961, Vol. 20, pp. 279–294.

1962

ILIFFE, J. K., and JODEIT, J. G. A dynamic storage allocation scheme, *Comput. J.*, Vol. 5 (Oct. 1962), pp. 200–209.

CODD, E. F. 'Multiprogramming', in Alt, F., *et al.*, *Advances in Computers*, Vol. III, Academic Press, New York, 1962, pp. 77–153.

GREENFIELD, M. N. FACT segmentation, *AFIPS Conf. Proc.* 21, Spartan Books, Baltimore, 1962, pp. 307–315.

1963

MCCARTHY, J., CORBATO, F. J., and DAGGETT, M. M. The linking segment subprogram language and linking loader, *Comm. ACM*, Vol. 6 (July 1963), pp. 391–395.

CRITCHLOW, A. J. 'Generalised Multiprocessing and Multiprogramming Systems', *AFIPS Conf. Proc.*, Vol. 24 (1963 FJCC), Spartan Books, Baltimore, 1963, pp. 107–126.

CONWAY, M. E. A multiprocessor system design, *AFIPS Conf. Proc.* 24, Spartan Books, Baltimore, 1963, pp. 139–146.

1964

DENNIS, J. B. Program structure in a multi-access computer, Tech. Rep. No. MAC–TR–11, Proj. MAC, MIT, Cambridge, Mass., 1964.

DESMONDE, W. H. *Real-Time Data Processing Systems: Introductory Concepts.* Prentice-Hall, Englewood Cliffs, N.J., 1964.

HAMLIN, J. E. A general description of the National Aeronautics and Space Administration real time computing complex, *Proc. ACM 19th Nat. Conf.*, Philadelphia, 1964, pp. 2–1 to 2–22.

1965

FANO, R. M. The MAC system: the computer utility approach, *IEEE Spectrum* 2 (Jan. 1965), pp. 56–64.

DENNIS, J. B. Segmentation and the design of multi-programmed computer systems, *J. ACM*, Vol. 12 (Oct. 1965), pp. 589–602.

COMFORT, W. T. A computing system design for user service, *AFIPS Conf. Proc.* 28, Spartan Books, Baltimore, 1965, pp. 619–626.

CORBATO, F. J., and VYSSOTSKY, V. A. Introduction and Overview of the Multics System, *AFIPS Conf. Proc.*, Vol. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 185–196.

CORBATO, F., and GLASER, E. *et al.* The Multic ssystem, *Proc. AFIPS* 1965 *Fall Joint Comput. Conf.*, Vol. 27 (Nov. 1965), pp. 185–196.

DALEY, R. C., and NEUMANN, P. G. A General-Purpose File System for Secondary Storage, *AFIPS Conf. Proc.* Vol. 27 (1965 FJCC), Spartan Books, Washington, D.C., 1965, pp. 213–229.

DENNIS, J. B., and GLASER, E. The structure of online information processing systems, *Information Systems Sciences: Proc. Second Conf.*, Spartan Books, Baltimore, 1965, pp. 1–11.

FORGIE, J. W. A time and memory-sharing executive program for quick-response, on-line applications, *AFIPS Conf. Proc.* 28, Spartan Books, Baltimore, 1965, p. 599–609.

GI ASER, E. L., *et al.* System Design of a Computer for Time Sharing Application, *AFIPS Conf. Proc.*, Vol. 27 (1965 FJCC), Spartan Books, Washington D.C. 1965, pp. 197–202.

MCCULLOUGH, J. D., SPEIERMAN, K. H., and ZURCHER, F. W. A design for a multiple user multiprocessing system, *AFIPS Conf. Proc.*, Vol. 28, Spartan Books, Baltimore, 1965, p. 611–617.

VYSSOTSKY, V. A., *et al.* Structure of the Multics Supervisor, *AFIPS Conf. Proc.*, Vol. 27 (1965 FJCC), Spartan Books Washington, D.C., 1965, pp. 203–213.

1966

ARDEN, B. W. *et al.* Program and Addressing Structure in a Time-Sharing Environment, *JACM*, Vol. 13 (Jan. 1966), pp. 1–16.

DENNIS, J. B., and VAN HORN, E. C. Programming Semantics for Multiprogrammed Computations, *Comm. ACM*, Vol. 9 (March 1966), pp. 143–155.

GOSDEN, J. A. Explicit parallel processing description and control in Programmes for multi and uni-processor computers, *Proc. AFIPS FJCC* (1966), Spartan Books, Washington, pp. 651–60.

MENDELSON, M. J., and ENGLAND, A. W. The SDS Sigma 7: A real-time time-sharing computer, *Proc. AFIPS FJCC* (1966), Spartan Books, Washington, p. 51–64.

SALTZER, J. H. Traffic Control in a Multiplexed Computer System. MAC. TR. 30. MIT, Cambridge, 1966.

WITT, B. L. The Functional Structure of OS/360: Part II, Job and Task Management, *IBM Systems Journal*, Vol. 5 (1966), pp. 12–29.

1967

COHEN, J. A use of fast and slow memories in list processing languages, *Comm. ACM.*, Vol. 10 (Feb. 1967), pp. 82–86.

BOBROW, D. G., and MURPHEY, D. L. Structure of a LISP system using two-level storage, *Comm. ACM* Vol. 10, pp. 155–159.