A MEMORY-MANAGEMENT UNIT FOR THE OPTIMAL EXPLOITATION OF A SMALL ADDRESS SPACE

C. BRON

Department of Applied Mathematics, Twente University of Technology, 7500 AE Enschede, The Netherlands

E.J. DIJKSTRA and S.D. SWIERSTRA

Department of Mathematics, State University of Groningen, 9700 AV Groningen, The Netherlands

Received 4 January 1982; revised version received 27 April 1982

Keywords: Address mapping, memory management, segmentation, shared address spaces

1. Terminology

We make a distinction between virtual memory and address mapping. Virtual memory is concerned with the mapping of a potentially large address space to a small (primary) memory. Address mapping (sometimes called memory management) is concerned with the mapping of a relatively small logical address space (e.g., limited to a 16 bit address) to a potentially large memory (e.g., using a 24 bit address). Memory management is in most cases used to locate the data (and the code) of several independent processes in a single large memory. This enables switching between these processes by simply switching the address map, thus circumventing slow backing store transfers.

With the advent of low cost MOS-memory devices, address mapping is a simple construct to exploit mass stores without having to upgrade the basic architecture of a processor, in which the size of the logical address space is a basic design decision.

2. Classical address mapping

The simplest form of address mapping is realized with a single relocation register, which contains the address in memory from which the current logical address space is contiguously mapped. Although address spaces may partially overlap with this form of address mapping it is difficult to make use of this form of sharing. In particular, if code and data are located in the same address space, this mapping makes sharing of code impossible. Since some form of sharing obviously must take place (in order to implement the communication and synchronisation between the processes) this crude mapping is insufficient.

Without going into the details of various mapping schemes, we briefly discuss the general form of the prevailing approach.

- Processors often provide (in their architecture) an instruction to switch between address maps (usually between user mode and kernel mode). This at least enables the operating system to be shared between processes. Transfer of data between the user address space and the kernel address space is done either by means of special instructions, or by mapping the address space as follows.
- Instead of mapping the address space as a whole, the address space is split up into a number of equally sized blocks, which can be mapped individually. Sharing can now be achieved by mapping blocks from several logical address spaces onto the same block of memory. This can be done at disjoint time intervals or simultaneously.
This block-wise mapping is in most cases achieved as follows. Each logical address is split in two parts, the most significant part (S), which is used as an index into an array of relocation registers (A) and the least significant part (D). The memory address (M) is now computed as follows:

\[ M = A[S] \cdot 2^K + D. \]

Note that if \( K \) equals the number of bits in D, the addition in this formula degenerates to a concatenation. The size of the addressable physical address space is determined by the number of bits in the elements of A and the value of K. The beginning of a block in physical memory is always at a \( 2^K \) boundary.

It need hardly be argued that sharing data by sharing large fixed portions of the address space may be quite uncomfortable.

Sharing by means of mappings as described above may give rise to all sorts of horrible overlay schemes. Furthermore, the placing of data together in one block will be based more on considerations of “fitting together in one block” than on “belonging together in one segment”. In general, since the block size is not dictated by logical arguments, part of the address space will remain unused and therefore the effective size of the address space, being rather limited to start with, is even further reduced.

The remainder of this paper deals with:
- The description of an address mapping unit which allows a linear address space to be split up into blocks of arbitrary size.
- The description of two ways in which this added flexibility may be used.

3. Flexible segmentation

Consider linear address spaces, where the addresses are numbered \( 0 \leq M < 2^W \), \( W \) being the width of a logical address.

We propose to split the logical address space into a number (L) of segments of arbitrary length by introducing segment boundaries \( A_i \), under the following constraints:

\[ A_0 = 0 < A_1 < A_2 < \cdots < A_i < \cdots < A_{L-1} A_L = M. \]

The address mapping unit contains a sufficient number of register pairs \( (A_i, B_i) \) and performs the following calculation on a given address \( X \).

For the single value of \( i \) satisfying

\[ A_i \leq X < A_{i+1} \]

the physical address \( M \) is calculated as follows:

\[ M = B_i \cdot 2^K + (X - A_i). \]

Thus the unit determines to which segment the address \( X \) belongs, and adds the relative address of \( X \) within the segment to a (shifted) address in the physical store.

For the memory management unit to be fast the address \( X \) has to be fed to all register pairs simultaneously. Furthermore it is possible to keep the value

\[ B'_i = B_i \cdot 2^K - A_i \]

instead of \( B_i \) in the B registers. When the comparison of the address \( X \) with the \( A \) registers is done in parallel with the addition of \( X \) to all the \( B'_i \), the memory address is available immediately after these computations. The selection of the result can easily be based on the change in the sign bits of the comparisons.

In the next section it will be shown how a modest number of segmentation registers (e.g., 4 to 16) is sufficient to support an attractive system structure. After all, it is not so much the number of registers in a classical memory management unit which is the restrictive factor, but the rigid segment boundaries within the address space.

4. Mapping tree structured systems

Consider a tree structured system, in which the nodes represent the individual processes, and the path from a node to the root represents the total address space of the process in that node. Thus the address space of a parent process is shared by all its descendants. Note that it depends upon the language, its scope-rules, protection facilities and the programs making up the system, what the actual visibility is of objects residing in this address space.

This method of sharing memory shows great
similarity to the organization within the Burroughs B6700/7700 series. The implementation of shared data structures (cf. UNIX pipes) could be implemented by introducing a process implementing the data structure. The child processes can now share the data structure in the parent process. With this system structure there is no need to locate objects in the operating system for them to be sharable.

If we single out any particular path from the root to a node, we may associate a stack with each branch passed (we know perfectly well how to map a stack into a linear address space). At every split along the path a new stack segment starts; the segments along the path are the obvious candidates from which, with the help of the mapping unit, our linear address space is constructed. In other words, the way in which a process at a leaf is instantiated and operates, is, from an addressing point of view, not different from a process in a single process environment on a single stack machine. Only when process switching occurs, i.e., when activity is delegated to another node, some address mapping registers must be altered.

Because the distinction in the architecture between shared and private data has disappeared, calls to the operating system can be ordinary procedure calls instead of calls having a switch in the address map associated with them. The basic operating system is formed by the process at the root of the tree.

Note, in particular, that processes closer to the root of the tree have access to data located in processes at a greater distance, if the addresses of these data have been passed as arguments to a call. There are no address space switches or mapping switches involved.

As a second use of our address mapping unit we would like to mention the implementation of a conventional Algol-like language on a machine with a small logical address space. Several data structures are involved in such an implementation:
- procedure incarnation and parameter stack,
- a stack containing linkage information,
- a heap,
- a runtime package,
- a code segment.

Usually the size of the fixed-size blocks in conventional address mapping units makes the mapping of these data structures onto such blocks very unattractive, especially because the sizes of these data structures are very different. It is here that our form of address mapping can economize on logical address space, because the size of the segments can be matched more exactly to the size of the data structures.

5. On the number of segmentation registers

If segmentation is used for the tree system, the number of segments will equal the degree of nesting with respect to process creation and will in general be small. A relatively small number of segmentation registers will do.

In the description we have focussed attention on those aspects of the architecture which make our form of address mapping different from other forms. It goes without saying that in an actual implementation of such an address mapping scheme all sorts of refinements are possible. As such we mention: presence bits, bits governing the access to the data, referenced bits, bits indicating that the data in a segment have been modified, etc.