Exploiting contiguity in a linear store.

Warning. In this text I have regularly followed the custom of underlining technical terms at their first occurrence in a context that is supposed to define them implicitly. This is a bad practice, and I apologize for having yielded to the temptation. (End of warning.)

My starting point is a classical von Neumann store consisting of 2^N locations, each of the same capacity, and each with its own address, the set of addresses being the numbers from 0 through 2^N-1. The value N is known as the address length. Access of a location --either for changing or for extracting its contents-- requires that the address of the location is supplied to the selection mechanism. I assume that the locations in store are randomly accessible.

The fact that the available addresses, besides being different from each other --a necessity because they have to identify locations uniquely in store-- , form the set of consecutive integers (from 0 through 2^N-1) is of great importance in combination with the circumstance that electronic equipment is very good at adding: it allows subscription. If m vector elements $a[i]$ (with $0 \leq i < m$) are stored in contiguous locations --i.e. locations with successive addresses-- selection of $a[i]$ can be implemented by computing the address of $a[i]$ by adding i to the address of $a[0]$; this subscription is assumed to be fast, and the time it takes is supposed to be independent of the value of i .

In view of the central position of the subscription we are entitled to attach great importance to the notion of contiguity. Because in a randomly accessible store each location is "as good as any other", we may even expect that the fact that the elements of a vector are stored in order in a set of contiguous locations will be of a much profounder significance than the actual position the vector occupies in store.

The fact that the vector's position as a whole is less significant than its being a vector has been obscured in the past by the high cost (in time) of moving a vector, a time which was proportional to the length of the vector. This high cost has resulted in a relative immobility, and in overestimation

of the significance of the actual position.

In order to do justice to this difference in significance between contiguity and position, we assume the availability of a storage shift operation. Indicating with M[i] the contents of location with address i, and with M' the store before the shift operation, the latter establishes

$$M'[i] = M[i+t]$$

for constant t and a contiguous range of i-values. For the time being I assume the storage shift operation only available for two values of t: t = + 1 (shift up) and t = - 1 (shift down). The shift operation is assumed to be a fast operation, in time independent of the length of the vector being shifted. Its availability is hoped to ease a number of the traditional storage allocation problems.

* * *

The use of storage shift operations may ease a number of storage allocation problems, it certainly introduces a new problem: it requires the introduction of a new, let me call it, "level of identification". It does so in the following manner.

At any moment in time the state of the computation is recorded in the current values of a number of variables; the actual number of variables may change as the computation evolves: upon block entry, new variables are introduced, upon block exit, existing variables disappear. In terms of storage usage: at any moment in time a number of locations is used to store the current value of currently existing variables, the other storage locations are free. If information is not shifted, each variable can be identified uniquely during its lifetime by the address of the location allocated to it upon its introduction. The flexibility to shift information, i.e. the possibility to reallocate variables during their lifetime, disqualifies the location address as identification for the variable. Hence the need for a new "level of identification".

One way of identifying variables is by the ordinal number of their creation --in very much the same way as is done in some political parties:

"party member nr.83" is then almost one of the founders. (In the Dutch Computer Society I used to be "nr.17"!)-- . When an old number is never reissued, this is an identifying "overkill", for we only need an identification mechanism distinguishing between any two concurrently existing variables; this overkill can be avoided by giving each variable upon creation "the lowest free number" say. These are very neutral techniques for generating an identification for each new variable created, so neutral that, later on, they give rise to high processing costs --usually in terms of large associatative stores or their simulations-- . The reason for the high price is that nothing can be deduced from contiguity of such numbers: from the existence of variables nr. 812 and nr.814, we can deduce nothing about the existence of variable nr.813.

I propose to explore an almost opposite technique for generating the new "level of identification": during its lifetime each variable will be identified by a so-called "global index train" of some length

$$c_1, c_2, \cdots, c_{n-1}, c_n \quad .$$

Contiguity in the nomenclature is intended to be reflected by the fact that the existence of a variable with the above global index train implies the existence of variables with global index trains

$$c_1, c_2, \cdots, c_{n-1}, i$$

for all values if i satisfying $0 \leq i \leq c_n$.

Note that not all variables need to have global index trains of the same length. At any moment in time, different variables must have different global index trains; if the global index train of A is a beginning subsequence of the longer global index train of B , the variable B will emerge as a component of the composite variable A .

* * *

In order to investigate the viability of such a proposal we have to investigate at least the following aspects:

1) how should a programming language implementation issue new global index trains upon "block entry", in general "when new variables are created"?
2) what algorithm will perform the general address calculation, i.e.

given the global index train of a variable, how is its address derived?

3) besides the actual global index train the general address calculation just mentioned has to process the "positional information" that has to be adjusted whenever information is shifted in memory; the question is how to represent this positional information such that

3.1) general address calculation is not unduly complicated

3.2) in case of information shifting the updating of the positional information is a well-defined and modest task.

4) how --when for 2), 3), and possibly to a certain extent for 1)-- proposals have been chosen, can additional hardware be used for speeding up the selection process?

* * *

I shall first deal with 2) and 3): they present a less open-ended problem than 1) and 4), and I do have a proposal that (even after 10 days of dreaming about it) strikes me as convincing. I shall first describe it in its purest form.

Denoting, as before, the contents of location with address i as $M[i]$, the address that corresponds to the global index train c_1 through c_n is in principle computed by the following algorithm:

```
i, b := 1, 0;
do i ≤ n → i, b := i+1, b + M[b+c_i] od        .        (1)
```

This two-line algorithm reflects the notion of so-called nested segments. A segment occupies in store a set of contiguous store locations and is a unit of contiguity. The final value of b after processing according to (1) a global index train of length n is called the base address of the corresponding segment which is a segment of order n. The total store itself is in this terminology the one and only, implicit, segment of order 0.

There are two types of segments, leaf segments and node segments. Leaf segments are the ones that are identified by a global index train that is not the beginning subsequence of a longer existing global index train; the others are called node segments.

The possible values of the variable b during execution of the address calculation (1) are called the segment base addresses. By inspecting (1) we see that 0 is by definition the segment base address of the only segment of order 0, and that M[0], M[1], ... , M[mc1] are the segment base addresses of the segments of order 1, if mc1 is the maximum value of the leading index c_1 of the global index train. In other words, the first mc1 locations of the segment of order 0 --together called the header of that segment-- contain what we can regognize as the mc1 descriptors of the mc1 segments of order 1. Those segments themselves are placed in the remaining locations of the segment of order 0. The segments of order 1 are all subsegments of the same segment of order 0. In exactly the same way, each segment of order i (>0) is a subsegment of one segment of order i-1, to be more precise: the segment with global index train c_1 through c_i is a subsegment of the (unique) segment with global index train c_1 through c_{i-1}; we call the latter the first's "supersegment".

Remark 1. Because different global index trains processed by (1) must lead to different final values of b, the subsegments of a segment occupy different locations in their supersegment. (End of remark 1.)

Remark 2. As far as (1) is concerned the order in which inside a segment its subsegments are allocated is immaterial. For simplicity's sake I assume that it is the same order as in which their descriptors are stored in the segment's header. If the descriptor also contains the length of the corresponding subsegment, this eases the administration of unused location between the subsegments, or between the header and the first subsegment or beyond the last subsegment. (End of remark 2.)

Inspection of (1) also explains why we have called the segment a unit of contiguity: the sum $b+c_i$ leads to a descriptor, then b + descriptor leads to the segment base address of one order higher.

The descriptors in the headers contain the "positional information" referred to above under 3). We can now answer the question mentioned under 3.2): if a segment is shifted up over 1 location --under the assumed initial presence of at least one free location beyond it in its supersegment-- its

descriptor should be increased by 1; if a segment is shifted down --again: under the assumed initial presence of at least one free location in front of it in its supersegment-- its descriptor should be decreased by 1. Algorithm (1) implies that when a segment is shifted, the adjustment of its descriptor is the only adjustment of positional information required.

Remark 3. Note that our convention of nested segments, where each segment has a defined length and a defined position in its supersegment, implies that each free location belongs to exactly one segment. (End of remark 3.)

* * *

The next thing we investigated was programming language implementation. At the start we had only two principles that seemed reasonably firm.

First principle. System information pertinent to each segment will be stored in the segment descriptor rather than in the segment itself.

A very simple example is provided by the segment length: we have to choose between storing the length of a segment in its descriptor or, say, in the segment's first location. The first principle says that it will be stored in its descriptor. I must admit that I have no strong further justifications for this first principle; I tried to find them, but none of the attempts at justification was anymore convincing than the first principle itself. Hence it seems more honest to stop this "petitio principii", and to postulate the First Principle openly as such.

Second principle. The addressing scheme considered will only interpret descriptors and all the information in descriptors will be relevant to this purpose; as a corollary: a leaf segment occupies a number of locations, the contents of which remain at this level uninterpreted and its descriptor contains no information about the interpretation thus abstracted from.

Suppose that at a certain block entry three single-length integers and two double-length integers are to be introduced, requiring together seven locations. We could have introduced all sorts of different kinds of leaf segments, one for each type. Then we would introduce separate leaf segments for the three single-length integers and the two double-length integers, and

the descriptors would then contain an indication of the type of information stored in the leaf segment. Such an arrangement, however, is ruled out by our Second Principle: in the example a leaf segment of seven consecutive locations will be introduced.

Again I must admit that I have no strong further justification for this principle. It can be viewed as an effort of separating concerns. In this report we are concerned with placing sequences of bits in location sequences, and I would like to treat that (if possible: the principles are tentative!) in isolation and independence of whatever "primitive data types" can eventually be distinguished in a full-blown language implementation.

For a while we thought that we would have two types of segments: uninterpreted leaf segments, and node segments containing a header and as many subsegments as descriptors in that header. But when we tried to use this, we encountered the following dilemma.

Suppose that at a block entry we introduce three single-length integers x , y , and z , and two vectors u and v of varying length. Because the lengths of u and v vary independently, we assume that both get their own subsegment with descriptors du and dv respectively. Originally we decided to allocate for x, y, and z a leaf segment with descriptor dxyz say. But then we have the dilemma that we can implement the block in two equally defensible (or indefensible) ways:

1) For the block incarnation we introduce a node segment with a three-descriptor header, containing dxyz, du , and dv . In this case the index chain identifying the sequence of storage locations allocated to xyz is one longer that necessary, the potential variation in size of the leaf segment for x, y, and z remains unexploited.

2) We really make the block entry a double block entry that increases the (static) block height with 2 instead of with 1 . In the outer block entry a leaf segment for x, y, and z is introduced, in the inner block entry a node segment with a two-descriptor header containing du and dv is introduced. Here the mixed introduction of scalar and vector variables gives rise to in increase of block height.

Both are equally viable and equally ugly. What we would like to do is to introduce one segment with descriptor dxyzuv, containing the (fixed size) "leafy" information x , y , and z and the two-descriptor header for du and dv (and the two corresponding subsegments). The conclusion is that it was too severe a constraint to confine uninterpreted information (such as x , y , z) to leaves, but that it must be allowed in nodes as well. A general node segment then consists of three areas (forgetting free locations for a moment)

1) uninterpreted information
2) header
3) subsegments (as many as there are descriptors in the header)

When the last two are empty, it only contains uninterpreted information and is identical to our original leaf; if the uninterpreted information is missing, it is in shape like the restricted node we originally considered. The original distinction between leaf segments and node segments seems to have largely disappeared: we are left with one type of segment, in which either the uninterpreted information or header+subsegment may be missing. This seems a definite improvement, but we get nothing free: the price to be paid is a more elaborate descriptor, which now contains four values, say:

ps (= segment position)	b+ps = address of first location of the segment
ns (= segment length)	b+ps+ns-1 = address of the last location of the segment
ph (= header position)	b+ph = address of the location of the first descriptor
nh (= header length)	b+ph+nh-1 = address of the location of the last descriptor

In the case that a segment is shifted up, the ps and ph in its descriptor have to be increased by 1 . In all cases we have $ps \leq ph \leq ph+nh \leq ps+ns$. A pure leaf is charaterized by $nh = 0$; if $ps = ph$, the uninterpreted information is missing.

In a segment ph - ps from its descriptor equals the number of locations allocated to the uninterpreted information. It is at this stage an open question whether we shall allow this difference to change during the existence of a segment with $nh > 0$, i.e. a non-empty header. If we do, the difference between a node segment and a leaf segment disappears still further, because then both can accomodate uninterpreted information of

varying size; it would, however, require a second type of shift operation, viz. a shift operation that does not shift a subsegment, but shifts a header (rquiring adjustment of ph only). If we don't, the difference between leaves and nodes is a little bit more marked, as then only leaves can accommodate uninterpreted information of varying size. The question is still open, I think that I have a mild, intuitive preference for the second choice; the reason is probably that then we can come away with one type of shift operations only.

It seems that the implementation of a single sequential program --like an ALGOL 60 program with the difference that array sizes are viewed as varying-- can now proceed in a fairly standard manner. With such a sequential process we associate what we might call a "stack segment". The --at this level!-- uninterpreted information of the stack segment is available --by way of "stack bottom", so to speak-- for the fixed amount of system information that need to be stored for each sequential process. Its header grows by one descriptor each time a procedure is called --and shrinks again upon return from a procedure-- ; the corresponding subsegments --we might call them "block segments" are introduced for the local variables of the procedure called: the uninterpreted information of the block segment is available for the local variables of the block, in the header of a block segment we introduce a descriptor for each local array. In contrast to stack segments, block segments will have constant header lengths. (Further decisions have to be taken about the allocation of anonymous intermediate result, actual parameters being passed, and return and further linking information; although important, these questions will not be pursued now.)

The important conclusion is that each block segment is identified by an index train consisting of the index train of the stack segment --which also serves to identify the process-- followed by one more index, which equals the dynamic depth. As long as parameter passing is confined within this sequential program, this last index --the dynamic depth-- suffices for the identification of block segments, and is the proper tool for block segment identification when, say, scalar variables are passed by reference. If procedures have access to global variables in the style of ALGOL 60, the display --or its equivalent-- would be used to translate the (static) block heights into the corresponding dynamic depths.

We encounter new problems, however, as soon as we try to accommodate a possibly varying number of such sequential processes --each with its own stack segment-- that, in addition, should be allowed to communicate with each other. An added complication --that is: a complication for me, while writing this text-- is that I would prefer to commit myself as little as possible about the, say, linguistic form in which several such communicating sequential processes are presented. I would like my treatment to be equally applicable to a co-routine organization of what is essentially view as a single-processor program, as to what is essentially presented as a set of concurrent processes on which in principle an equal number of processors could be engaged. In view of this desire not to commit myself in either direction I propose not to discuss in this report how the progress synchronization between various sequential processes is going to be organized, with but one exception, viz. my one and only "model" for process creation and destruction.

In a sequential program may occur a so-called "parallel compound", say

[S1 || S2 || S3] .

(Here, following C.A.R.Hoare, I have used square brackets as statement brackets and two vertical bars --as opposed to a semicolon-- to separate the components of a parallel compound.)

The beginning of the execution of the parallel compound coincides with the beginning of the execution of all its parallel components --in the above example S1 , S2 , and S3 -- , termination of the parallel compound is the termination of all its components. For the time being we confine ourselves to parallel compounds with a finite number of components that is known when its execution starts.

In a case like the above three new stack segments have to be created at the start of the parallel compound, and, again, it is an implementation decision what index chain to assign to them. My proposal is to introduce the three stack segments, corresponding to S1 , S2 , and S3 respectively, as subsegments of the stack segment corresponding to the sequential process in which the parallel compound is executed. More precisely:

let C be the index chain identifying (the stack segment of) the process that is about to split into three, and let i be the current dynamic depth --header length-- such that a block entry would cause the creation of a block segment with index chain "C, i" ; then the entry of the parallel compound

[S1 || S2 || S3]

would cause the creation of three stack segments --note: not block segments!-- with index chains "C, i", "C, i+1", and "C, i+2" respectively.

The rationale behind this proposal is exactly the same one as the one that would have assigned the index chain "C, i" to the newly created block segment if, instead of the parallel compound, a normal block would have been entered: contiguity of used values of the last index is maintained and, perhaps more important: "C, i", "C, i+1", and "C, i+2" are by definition available! This is in strong contrast to the situation in which the start of the three-fold parallel compound would have implied the acquiring of three stack segments from a common pool of free ones. In the latter arrangement two such (possibly) concurrent splits would almost certainly need to be implemented under mutual exclusion in time, and that is exactly the kind of interdependence that with the nested segments I seek to avoid.

This proposal definitely introduces a phenomenon that otherwise perhaps could have been avoided: our version of the so-called "cactus stack" now leads to stack segments --and hence to sequential processes-- identified by index chains of different lengths.

I fear that a severe implementation problem presents itself when we are requested to cope with what we might call a "recursive split", as occurs when from the components of a parallel compound within the body of a procedure that same procedure may effectively be called again. The possible depth of the cactus stack is then in principle as unlimited as the depth of a stack in the case of the traditional implementation of a sequential recursive procedure. In that traditional case, each activation is characterized by its depth, i.e. a scalar, which may become large, but remains a scalar. The depth of the cactus, however, equals the length of the index chain of the stack segments in its top, and the manipulation of such long index chains is definitely unattractive. Although there is conceptually nothing wrong with

the recursive split, it has another nasty consequence: if the procedure can be called from more than one component of its internal parallel compound, the number of stack segments to be identified and allocated may grow as an exponential function of the depth. So there is some justification to chicken out and to restrict ourselves to exploring what can be done when a recursive split is excluded or by other means it can be guaranteed that the maximum depth of the cactus stack remains under a modest upper bound.

Under that restriction we can assume each sequential processor equipped with enough registers, i.e. one for each index --and probably also the corresponding segment base address-- of the global index chain identifying (the stack segment and) the sequential process currently executed by it. They would give access to the "dynamically enclosing" stack segments; they are in the cactus stack the ones on the path from the current stack down to the root of the cactus, and along that path each of them is suitably identified by one of the integers from 0 through n-1 (if the global index train of the current stack segment is n indices long): 0 for the root of the cactus stack and n-1 for the current stack segment. I propose to call these integers "ranks" of these stack segments. (A split in a process with stack segment of rank r creates a number of new stack segments of rank r+1 .)

As long as direct contact to (block segments in) other stack segments is confined to "dynamically enclosing" stack segments, each of these segments is sufficiently identified by its rank; the display elements --see EWD653-8, last paragraph-- would then contain, besides the dynamic depth identifying the block segment with a stack segment, the rank of that stack segment in order to identify the latter.

Note. The restriction mentioned at the beginning of the preceding paragraph seems very strong, and I am not entirely happy with it. I feel, however, comforted by the consideration that, without further synchronization constraints, the dynamically enclosing stack segments are the only ones the existence of which can be guaranteed from within the current process. (End of note.)

Remark. We remind the reader that the text of a procedure does not fix the dynamic depth of the block segment corresponding to its call. In complete

analogy, the text of a procedure does not fix the rank of the process from which it is called, and if the procedure body contains a parallel compound the rank of the component processes is not fixed by the procedure text: their rank is by definition one higher than the rank of the process that called the procedure. (End of remark.)

With the above I seem to have achieved two further design goals that, although constantly at the back of my mind, have not been mentioned explicitly yet, viz.

1) text does not reflect the rank of the sequential process in which it will be executed; this is, of course, necessary if procedures of a library are to be generally available--i.e. at different ranks-- .

2) the efficiency with which a process is executed need not depend on the rank at which it is executed.

When two processes should communicate with each other, the global index chain identifying the one cannot be equal to the beginning of the global index chain of the other: if this were the case the one of the highest rank --being a (sub)component of the other-- would be the only active one of the two. When two processes have to communicate with each other this communication is visualized as taking place via the highest rank stack segment that contains the stack segments of the two processes. This choice is a natural one, because it is the smallest environment the existence of which can be guaranteed when both communicating processes exist. (And, as long as we don't state more precisely what is meant by "visualized as taking place via" the proposal seems harmless.)

* * *

Having reached this stage I have the feeling that the further exploration of point 4) --what further hardware assistance is feasible?-- should not be done now. I doubt whether I am the appropriate person to do so, and in any case I think that these investigations should be postponed until more people have looked at and commented upon the above --may I call it "Conceptual"?-- design. I know that various people are thinking about programming languages of the "Communicating Sequential Processes" type, and it would be nice if they could confront their ideas with the above outline of a machine structure.

Other things such a confrontation might elucidate are the viability of the separation of concerns sketched at EWD653 - 6, second paragraph and the proper way of storing anonymous intermediate results and actual parameters. Almost certainly such confrontations will raise more specific questions concerning synchronization and message passing. Also the question how "monitors" fit into this picture should be answered.

The writing of the last part of this report --in particular pages 9 through 12-- caused me trouble enough and took much more time than anticipated. (Any suggestions as to how to improve the presentation of that part of the design would be most welcome!) In the meantime it seems better to conclude this report and distribute it in its imperfect and incomplete state.

Plataanstraat 5
5671 AL NUENEN
The Netherlands

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow