On-the-fly garbage collection: an exercise in multiprocessing.

(After careful consideration of a wider class of problems, A.J.Martin and E.F.M.Steffens selected and formulated the following problem and did most of the preliminary investigations. I arrived at its solution during a discussion with the latter, W.H.J.Feijen and M.Rem. It is a pleasure to acknowledge their share in its discovery.)

With the emerging advent of multiprocessor installations, all of us have discovered that it is by no means obvious how a great number of processors should be engaged on a single task. One line of attack has been --and still is-- to try to discover problems that can be solved by a great number of concurrently active processors; on the one hand we are inspired and encouraged by some spectacular results, on the other hand it is somewhat discouraging that these techniques are always only applicable thanks to the rather special nature of the specific problem to be solved. The other line of attack is inspired by the observation that in any large scale computer installation today, a considerable amount of time of the (general purpose) processor is spent in "operating the system" and the question, therefore, emerges to what extent these "special purpose" activities inside a rather general purpose installation can be done concurrently with the execution of users' programs. Because the more intimate the interference, the harder the organization of the cooperation between the concurrent processes, the problem of garbage collection was chosen as one of the most challenging --and, hopefully: most instructive!-- target. Whether the following solution is of any economic significance is beyond the scope of this report: for the time being it suffices to be fascinated by its existence.

In the traditional LISP-environment the data structure to be stored at any moment is a directed graph in which each node has at most two outgoing edges. (By introducing a virtual target, called NIL, each node can always be given exactly two outgoing edges, a left-hand edge and a right-hand edge.) As a result the storage requirements for each node can be regarded as constant in time and equal for each node. The whole data structure has one node --called: the root-- with a constant place in memory; at any moment only those nodes that via the edges can be reached from the root are significant for the progress of the computation. The computation exists essentially of two operations: either replacing for a node one of its outgoing edges by one with (another)

existing node as its target, or by one pointing to a new target, i.e. a new node is to be added to the data structure. In the latter case the number of nodes of the data structure is increased by one: the new node is taken from the so-called "free list" --i.e. a linked list of node locations that are currently not used for storing a node of the data structure--. In the first case the number of nodes either remains constant or it decreases by a rather unpredictable amount: the removed edge may have been the last connection from the root to a possibly elaborate subgraph. It is the purpose of the so-called garbage collector to detect such disconnected and therefore obsolete nodes and to append them again to the free list.

In classical LISP-implementations the computation proceeds until the free list is exhausted (or nearly so). Then the LISP-computation comes to a grinding halt, during which the central processor is devoted to garbage collection, i.e. starting from the root the transitive closure as given by the current edges is established and all the nodes in memory outside this transitive closure --i.e. not reachable from the root and therefore obsolete-- are appended to the free list, after which operation the LISP-computation can proceed. The minor disadvantage of this arrangement is the central processor time spent on the collection of the garbage, its major disadvantage is the absolute unpredictability of these garbage collecting interludes, which makes it virtually impossible to design such a system so as to meet real time requirements as well. It was therefore tempting to investigate whether a second processor --called "the collector"-- could collect garbage on a more continuous basis, concurrently with the activity of the other one --for the purpose of this discussion called "the mutator"-- which is dedicated to the LISP-computation proper. Two additional constraints have to be obeyed: the (microscopic) interference between collector and mutator should be minimal --i.e. no highly frequent mutual exclusion of elaborate activities-- and, also, the overhead on the activity of the mutator (as required for the cooperation) should be kept as small as possible.

A certain amount of overhead for the mutator, alas, is unavoidable. Suppose that nodes A and B are permanently connected to the root via a constant set of edges, while node C is only connected to the root via an edge from A to C . Suppose furthermore that from then onwards the mutator performs with respect to C repeatedly the following four operations:

1) make an outgoing edge from B pointing to C
2) make the edge from A to C pointing to another node
3) make an outgoing edge from A pointing to C
4) make the edge from B to C pointing to another node.

The collector, which observes nodes one at a time, will discover that A and B can be reached from the root, but never needs to discover that C can be reached as well: while A is observed, C may be connected via B and the other way round.

Suppose that all nodes may have one of four colours: white, grey or black for nodes in the data structure and green for all nodes in the free list. (Whether the colour green is necessary is a question that I leave open: it certainly eases the description.) Suppose that all nodes in the free list are green and all other nodes are white. Collector and mutator will now start to colour nodes that can be reached from the root black, but in general this will happen via grey as intermediate colour. They do so, observing two rules:
a) nodes will only darken monotonically (where green is regarded as light as white)
b) no edge will ever point from a black node to a white one.

The mutator will act as follows. If it adds a new node to the data structure --i.e. takes a green node from the free list-- the mutator will make the source node of the new edge grey if it has observed it to be white and will leave the colour of the source node of the new edge unchanged if it has observed it to be grey or black; it changes the colour of the target node of the new edge from green to black. If the mutator makes an edge pointing to a node already in the data structure (i.e. currently pointed at by another edge), it will subject both source and target node of the new edge to the same treatment: make it grey if it has been observed to be white, otherwise leave its colour unchanged. For an increase of the number of grey nodes as a result of mutator activity, the presence of at least one white reachable node is therefore a necessary condition.

The collector starts by making the root grey. It the looks for grey nodes: for each grey node it inspects its two successors (i.e. target nodes of its outgoing edges): if a white successor is found, it is made grey, other-

wise its colour is left unchanged; when both its successors have thus been processed, the originally grey node --which could be its own successor!-- is made black.

Note 1. Several times we have said "if the node has been observed to be white it will be made grey, otherwise its colour will be left unchanged". We have intentionally avoided to say if its colour has been observed to be grey (or black) its colour will be made grey (or black). In that case the mutator could observe a grey node and make it grey, just after the collector has made it black, thus undoing the collector's activity. (End of note 1.)

Note 2. We have assumed that the observation of a node colour will never lead to the observation "white" while the other partner changes it from grey to black. (End of note 2.)

Note 3. The collector inspects the successors of a node observed by it to be grey. The mutator may simultaneously change the outgoing edges of that node. It is assumed that the collector will be directed towards either an old or a new successor and everything is safe provided that the mutator first adjusts the colours of the end nodes of the new edge and only places the new edge afterwards. (End of note 3.)

Besides (possibly) belonging to the data structure or the free list, the nodes are linearly ordered by their order in store. The collector can therefore inspect them all, in cyclic order, say. When during a cyclic inspection of all nodes, the collector has found no grey node, all reachable nodes (and possibly more, viz. nodes that have been reachable in the past) are black --the proof of this statement is not fully trivial-- and the mutator can create no new grey nodes anymore. Any nodes now white, will remain white: the collector can therefore add the now white nodes to the free list (by colouring them green etc.). After having done so, the nodes in the free list are green and all other nodes are black. In synchronism --but this is a relatively infrequent occurrence-- collector and mutator now invert their white/black interpretation (grey remains grey and green remains green) and the game starts all over again.

To fill in the further details --such as arranging the free list as

as first-in-first-out list so as to make simultaneous extension by the collector and consumption by the mutator possible, etc.-- and to prove that it is all safe and sound requires great care, but it can be done. (I know even people, who could certainly do it, such as Alain J.Martin of Philips Research Laboratory, Leslie Lamport of Massachusetts Computer Associates Inc. and probably also Severo M.Ornstein of Bolt, Beranek and Newman Inc. or one of his colleagues.)

* * *

As said in the introduction, a claim of economic significance is not made. The significance of the above solution lies in the fact that it displays a way of non-trivial cooperation between loosely coupled processes which differs rather radically from the usual communication via messages or mutually exclusive access to common variables. As the reader will have realized, the monotonicity argument (rule a) is as essentail as the invariance (rule b). Finally, in order to enable to collector to detect termination, it is essential that the mutator colours green nodes immediately black (as far as rules a and b are concerned, grey would have been permissible as well). Isn't it fascinating?

2nd April 1975

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow
Plataanstraat 5
NUENEN - 4565
The Netherlands