Fast Fits

New methods for dynamic storage allocation

(Summary of paper to be published in ACM Transactions on Computer Systems)

C. J. Stephenson

Thomas J. Watson Research Center
Yorktown Heights, N.Y.

Introduction

The classical methods for implementing dynamic storage allocation can be summarized thus:

First Fit and Best Fit

The available blocks of storage are linked together in address order. Storage is allocated from the first available block of sufficient length, or from a block with the minimum excess length. Storage can be allocated or released in multiples of two words. In the long run, if numerous pieces of storage of more-or-less random lengths are allocated and released at more-or-less random intervals, the storage becomes fragmented, and a number of uselessly small blocks develop, particularly near the beginning of the list. Although these fragments usually comprise a small proportion of the storage (typically around 10 per cent), a lot of time can be wasted chaining through them.

Buddy Methods

Here the task of managing the storage is reduced in size by constraining the way in which the storage can be divided up, e.g. into blocks with lengths which are powers of 2. This eliminates chaining through long lists of uselessly small blocks; on the other hand, space is wasted in rounding up the length requested to an allowable size, and typically about 40 per cent more storage is required to satisfy the same allocations than when using First Fit or Best Fit.

The methods presented in this paper are externally compatible with First Fit and Best Fit, and require roughly the same amount of storage for a given sequence of allocations. They use, however, a completely different internal data structure, one effect of which is to reduce the number of blocks that have to be visited to perform a typical allocation or release operation. These new methods exhibit roughly the same space performance as First Fit, and a time performance which falls between those of First Fit and Buddy.

The data structure

Instead of the available blocks being chained in address order (as in First Fit and Best Fit), they are chained in a tree in which for any node S:

address of descendants on left (if any) < address of S < address of descendants on right (if any);

length of descendants on left (if any) ≤ length of S ≥ length of descendants on right (if any).

An example is shown in Fig. 1.

Structures of this general form are described by Vuillemin in the April 1980 issue of CACM. He calls them 'cartesian' trees.

In the full version of the present paper, algorithms are presented (a) for inserting a node into such a tree, (b) for deleting a node, (c) for promoting a node (to accomodate an increase in length), and (d) for demoting a node (to accomodate a decrease). All of these operations involve visiting O(D) or fewer nodes, where D is the depth of the tree.

Using these algorithmic tools, it is possible to allocate a piece of storage from any node of sufficient length, and to demote the remainder of the node (if any), or to delete the node (if it is all used). Likewise, it is possible to insert a piece of storage being released as a new node, or to coalesce it with its neighbours in the tree (if it has any). In principle, coalescing a piece of storage with its neighbours involves first finding the neighbours, and then performing a promotion (if there is one neighbour) or a promotion and a deletion (if there are two); but in practice the search and the other operations can be combined so that usually only one descent through the tree is necessary.

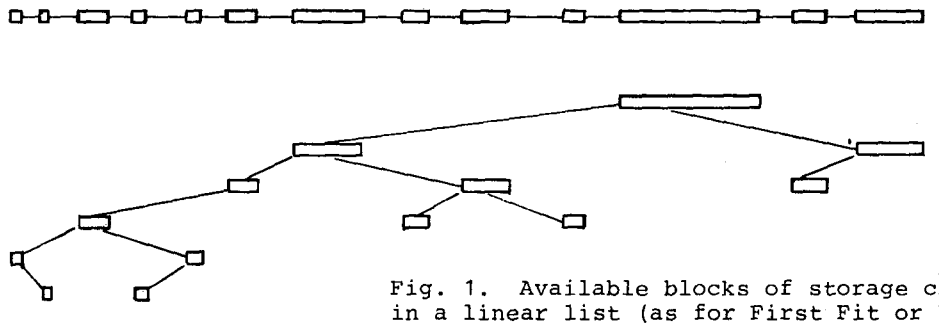

Fig. 1. Available blocks of storage chained together in a linear list (as for First Fit or Best Fit), and in a cartesian tree (as for Leftmost Fit or Better Fit). Storage addresses increase to the right.

It is possible to preserve the two-word granularity of First Fit and Best Fit, even though, using the tree, it is necessary to accomodate two pointers in each node in addition to length information. The nicest way of doing this is to store the length of a node in the same place as its address, i.e. in its paternal node, and to omit the length fields when the paternal node contains only two words.

Provided the tree does not become grossly unbalanced, this structure enables allocation and release operations to be performed in time O(log N), where N is the number of available blocks at the time of the operation. It also lends itself to speedy allocation policies, since a node of adequate length can be found without inspecting any nodes of inadequate length.

It should be noted, however, that the tree does not necessarily maintain good balance, and in general it cannot be balanced explicitly, since the position of each node, relative to the other nodes, is determined both horizontally and vertically. Whether the structure will in practice be well balanced therefore depends upon the actual pattern of allocation and release operations, together with the allocation policy.

Allocation policies

In certain situations, programs can request the longest piece of storage available, or can request a piece that is longer than any of the available blocks. This exposes one of the nastier aspects of First Fit and Best Fit, since the entire list must be searched. The methods presented here avoid this problem completely, for the longest node is always at the root, and its length is therefore available immediately.

Now consider the case in which the length requested does not exceed that of the root. How should a node be selected for allocation? Clearly, it would always be possible to select the root itself (and demote what remains, if necessary), but in general this would needlessly fragment the larger chunks of storage. There are at least two policies which seem more attractive, and which I have dubbed Leftmost Fit and Better Fit, as follows.

Leftmost Fit

This method selects the leftmost node of sufficient length. For a given sequence of allocation and release operations, the pieces allocated are identical to those that would be allocated if the same pool of storage was managed by First Fit, i.e. they have the same addresses and lengths. Consequently the fragmentation properties are also identical, and are known to be quite good. Since, however, allocation is always from the left-hand side of the tree (or in the limiting case from the root), the tree will tend to become unbalanced.

Better Fit

This method selects a node by descending the tree, from the root, so that at each decision point the better fitting son is chosen (i.e. the shorter one, provided it is long enough, or the longer one, otherwise). The descent stops when both sons are too short. This tends to preserve the larger nodes until they are really needed, and avoids the primary source of imbalance in Leftmost Fit.

Performance studies

The new methods were compared in performance with First Fit, Best Fit and the binary Buddy method, using randomly selected lengths and lifetimes for the allocated pieces of storage. The more important results can be summarized as follows.

The new methods do not offer any advantage, compared with First Fit or Best Fit, when only a few hundred pieces of storage are allocated. When however the storage pool is large, and there are several thousand pieces allocated simultaneously, there is a substantial benefit.

In one of the experiments, for example, there were (in the steady state) about 10,000 pieces of storage allocated, with a characteristic length of 100 words. Using Leftmost Fit, an average of 27 nodes were visited per allocation or release operation, compared with 1850 for First Fit and about 925 for Best Fit. Even after allowing for the

greater complexity of Leftmost Fit, this represents a big improvement in terms of the number of instructions executed.

In the experiments, Better Fit required about the same amount of storage as Leftmost Fit, and involved visiting even fewer nodes (because the tree was better balanced). There is however some evidence that, contrary to expectations, Better Fit may in certain realistic situations require significantly more storage than Leftmost Fit, and Better Fit should therefore be viewed with some caution.

Compared with any of these, the Buddy method performed well in terms of instructions executed, and badly in terms of the amount of storage required.

It may be noted that the Buddy method is not equivalent in function to the other methods, since it cannot support piecemeal release of an area, i.e. a piece of storage that is supplied in one allocation operation must be released as a unit. Whether this matters in practice depends upon the application.

Finally, by way of warning, it should be observed that since the performance of Leftmost Fit and Better Fit depends upon the shape of the cartesian tree, it is possible to find sequences of allocation and release operations which result in gross lack of balance, and consequently give rise to execution times which are no better (or even worse) than those of First Fit. Fortunately such patterns have not been observed in practice.

Acknowledgements

I thank the University of Hong Kong for providing computer facilities for testing these ideas in 1981, while I was a visitor there; and my colleages at IBM Research for good suggestions on the resulting paper.