A SIMPLE VARIANT OF THE BOUNDARY-TAG ALGORITHM FOR THE ALLOCATION OF COROUTINE ENVIRONMENTS

David R. HANSON
Department of Computer Science, The University of Arizona, Tucson, Arizona 85721, USA

Received 6 September 1975, revised manuscript received 6 November 1975

Storage allocation, coroutines, programming languages

1. Introduction

This note describes a simple variant of the boundary-tag algorithm [4] for the dynamic allocation of storage. The technique has been developed and successfully implemented for the allocation of coroutine environments in the Sil/2 programming language [2]. Sil/2 is a high-level system programming language used primarily for the construction of transportable interpreters. Currently, Sil/2 is being used for the implementation of an experimental language called SL5 [1,3]. One of the reasons for choosing Sil/2 was the desire to support a reasonably efficient implementation of coroutines.

An environment in Sil/2 is a variable-size block of storage that contains the values of the local variables for a particular instance of a coroutine. A variable can contain the address of an environment as its value, and such addresses are indistinguishable from positive integers. Thus the allocation policy must not cause the relocation of an active environment since the location of pointers to other environments cannot be determined. In the initial implementation of Sil/2 [2], allocation was performed in a stack-like fashion. Although this is very fast, it is inappropriate for the allocation of environments for coroutines. The standard stack method works well for the allocation of environments for recursive functions because the "lifespan" of an environment is well-defined. The lifespan of a coroutine environment, however, is usually unpredictable. Stack-like allocation gives an upper bound on the amount of storage required for coroutine environments, but does not allow storage to be reused unless blocks are allocated and returned in a first-in, last-out manner.

The free list scheme described here is based on the observation that the distribution of environment size during the execution of an SL5 program is semi-independent of the particular program. This distribution, obtained at runtime, is referred to as the *dynamic* distribution. The declarations of the routines in the Sil/2 source program define a *static* distribution of environment size, which can be obtained at compile-time. It has also been observed that the dynamic distribution of environment size is essentially the same as the static distribution. This is illustrated in fig. 1. Squares indicate the static distribution and circles are values obtained from empirical measurements of several SL5 programs.

These observations suggest that a simplified version of the boundary-tag method described by Knuth [4] can be used without significant loss of efficiency. The simplifications are that adjacent available blocks are not coalesced when the storage for an environment is returned and that the free list is not ordered by address. This makes the process of returning a block to the pool of available blocks trivial and very fast.

It is assumed that after the program reaches a "steady state" the distribution of blocks on the free list is approximately the same as shown in fig. 1. In other words, when the program reaches a state in which the rate of allocation is approximately equal to the rate of liberation, the distribution of available blocks tends to reflect the distribution of allocation

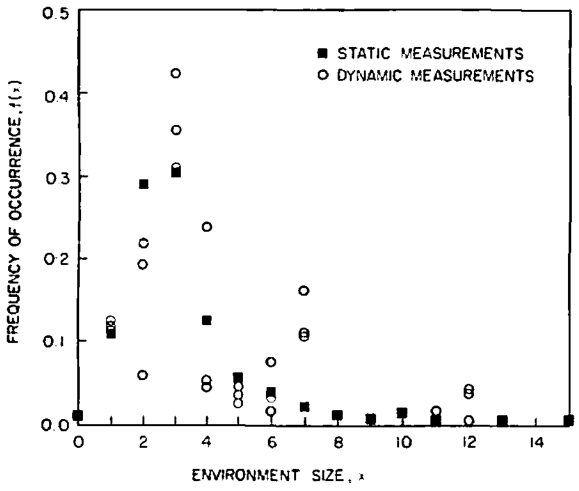

Fig. 1. Static and dynamic distributions of coroutine environment size.

requests. Therefore, for the majority of requests, allocation is quite fast and requires the examination of only a small portion of the free list.

2. Allocation and liberation

The storage for a coroutine environment is allocated from a fixed-size contiguous region of memory called a *vector*. The available blocks in a vector are linked together in a singly-linked free list. An available block is characterized by its first word containing the size of the block and its second word containing the address of the next block on the free list. These fields are referred to as the *size* and *link* fields respectively. A block housing an active coroutine environment is characterized by the *size* field containing the negative of the length of the allocated block.

The head of the free list is the link field of an active block that occupies the first two words of a vector. The last two words of a vector form a permanently active block whose presence simplifies the coalescing procedure (see the Appendix). Blocks are allocated beginning at the third word of a vector. Initially, the entire vector, except for the first and last two words, is composed of one available block.

Thus, at any time during execution, a vector is comprised of a number of self-identifying active environments and available blocks, the latter being linked together in a free list.

The *first fit* method [4] is used for the allocation of a block for a coroutine environment. The free list is searched in a linear fashion for the first block whose size is greater than or equal to the amount of the request. A large block is split into two smaller blocks only if the amount in excess of the request is large enough to accommodate a coroutine environment.

If an allocation request cannot be satisfied, a linear sweep of the vector is performed coalescing adjacent available blocks and simultaneously reconstructing the free list. This is possible because the blocks in the vector contain the information necessary to determine whether they are active or available.

Liberation is accomplished by simply inserting the block at the beginning of the free list and negating the contents of the *size* field.

Algorithms for allocation and liberation, written in PASCAL, are given in the Appendix.

3. Discussion

Initial experience with this simple technique indicates that it is well-suited to the particular application of allocating storage for coroutine environments. Execution times tend to be slightly longer than times measured using the stack-like allocation scheme. The stack method, however, can only liberate blocks that are used in a stack-like fashion. Heavy use of coroutines results in a significant number of blocks that cannot be reused. Consequently, the stack method requires that the vector be substantially larger than is necessary for the simplified free list method.

Results also indicate that the additional overhead in execution time can be attributed primarily to the time required to coalesce available blocks. As long as the vector does not need to be coalesced, allocation is quite fast. The number of coalescings required depends on the size of the vector and the point at which a large block is divided into two smaller blocks during allocation. In an initial version of this allocation scheme, a large block was divided if the excess was large enough to accommodate a *minimum-size* environment. This tended to proliferate small blocks, thereby distorting the expected distribution shown in fig. 1 and requiring that the vector be coalesced more often.

By requiring that the excess be large enough to accommodate an *average-size* environment, the number of coalescings may be significantly reduced.

For example, in programs that caused the allocation of 4000 blocks from a 512 word vector, the average length of the free list search was only 4 blocks. The vector had to be coalesced 8 times when large blocks were divided if the excess was enough for a minimum-size environment. When the cutoff point was increased so that the excess had to be enough for an average-size environment, only 3 coalescings were required. In addition, this modification also resulted in shorter searches of the free list; since the free list contained a large number of blocks of the average size, the request was often satisfied by the first block on the list.

The distribution shown in fig. 1 is marred by a distinct anomaly at $x = 7$. This is known to be due to the heavy use of a single routine whose environment size is 7 words rather than a large number of routines with an environment size of 7 words. This single routine, BLOCK, is the basic SL5 storage allocation routine. Thus the measurements provided not only a basis on which the environment allocation policy was chosen, but aided in identifying routines that could be revised to favor the chosen policy.

Acknowledgements

I would like to thank Dianne Britton, Douglas Brotz, Frederick Druseikis, Richard Orgass, and Ralph Griswold for numerous stimulating discussions concerning this subject. I would also like to express my appreciation to David Gries for a careful review of the manuscript and for suggesting the form in which the algorithms in the Appendix are presented. This work was supported by the National Science Foundation under Grant DCR75-01307.

References

[1] D.E. Britton, F.C. Druseikis, R.E. Griswold, D.R. Hanson and R.A. Holmes, Procedure referencing environments in SL5, to be presented at the Third Ann. SIGACT-SIGPLAN Symp. on the Principles of programming languages, January 1976.

[2] F.C. Druseikis, The design of transportable interpreters, SNOBOL4 Project Document S4D49, The University of Arizona, Tucson, February 1975.

[3] R.E. Griswold and D.R. Hanson, An overview of the SL5 programming language, SL5 Project Document S5LD1, The University of Arizona, Tucson, 1975.

[4] D.E. Knuth, The art of computer programming, Volume 1: Fundamental algorithms, sec. ed. (Addison-Wesley, Reading, Mass., 1973) section 2.5.

Appendix

The algorithms given in this appendix are written in PASCAL and assume the following declarations.

```
type vector = array [0 . . M] of integer;

const  minsize = 3; {cutoff point for dividing a large block}
       avail   = 0; {offset in vector of head of free list}
       first   = 2; {offset in vector where blocks are allocated}
       size = 0; {offset in block of the size field}
       link = 1; {offset in block of the link field}

var v: vector;
```

A vector is initialized as follows.

```
v [avail + size] = -2
v [avail + link] = first
v [first + size]  = M - 4
v [first + link]  = 0
v [M - 1] = -2
v [M] = 0
```

Blocks are allocated by the function *allocate* that returns the address of the first word of the allocated block and sets the *size* field to the negative of the length of the block. The function *allocate* calls *coalesce* if the vector must be coalesced in an attempt to satisfy the request. The procedure *coalesce* reconstructs the free list and determines the size of the largest available block. An error is signaled if the largest available block is too small to satisfy the request. Procedure *free* is called to return a block to the free list. It also negates the contents of the *size* field so that the block can be identified as available by *coalesce*.

In the comments, the notation $v[i\,..\,]$ denotes the block $v[i\,..\,v[i + \text{size}]]$.

```
{Find a free block v[p . . ] of size n or larger, allocate it, and return its index p. If this is not possible, signal an
error.}

function allocate (var v: vector; n: integer): integer;

var   q, {all free list blocks up to v[q . . ] have size < n}
      p, {p = v[q + link], index of free block following v[q . . ]}
      k: {(size of v[p . . ]) − n}
         integer;
      present: boolean; {indicates the size of v[p . . ] is >= n}

begin
   {Find the index of free block v[p . . ] with size >= n, letting v[q . . ] be its predecessor. Signal an error if no
   such block exists.}
      q := avail; present := false;
      repeat
         p := v[q + link];
         if p = 0 then
            {coalesce free blocks and initialize for another attempt}
               begin coalesce (v, n); q := avail; p := v[q + link] end;
         if v[p + size] < n then q := p else present := true
      until present;
{take at least n locations of v[p . . ] as the new block}
   k := v[p + size] − n;
   if k < minsize
      then v[q + link] := v[p + link]
      else begin v[p + size] := k; p := p + k; v[p + size] := n end;
{allocate the block and mark it active}
   allocate := p; v[p + size] := −v[p + size]
end {allocate};

{Coalesce the free blocks of v and reconstruct the free list. Signal an error if there are no free blocks of size
>= n.}

procedure coalesce (var v: vector; n: integer);

var   s, {size of largest free block processed thus far}
      p, {index of block currently being processed}
      k: {all adjacent blocks from v[p . . ] up to v[k . . ] are free}
         integer;

begin
   p := first; s := 0; v[avail + link] := 0;
   while p < M do
      if v[p + size] < 0
         then {skip active block v[p . . ]}
                  p := p − v[p + size]
         else begin
               {add available block v[p . . ] to free list}
                  v[p + link] := v[avail + link];
                  v[avail + link] := p;
               {find end index k − 1 of last adjacent free block}
                  k := p + v[p + size];
                  while v[k + size] > 0 do k := k + v[k + size];
               {coalesce following adjacent blocks into v[p . . ]}
                  v[p + size] := k − p;
                  s := max(s, k − p);
               p := k − v[k + size]
            end;
   if s < n then signal error
end {coalesce};

{Put block v[p . . ] back on the free list and mark it available.}

procedure free (var v: vector; p: integer);

begin
   v[p + link] := v[avail + link];
   v[avail + link] := p;
   v[p + size] := −v[p + size]
end {free};
```