```
From: wmb@MITCH.ENG.SUN.COM
Newsgroups: comp.lang.forth
Subject: Forth memory allocator
Message-ID: <9008101347.AA02701@ucbvax.Berkeley.EDU>
Date: 10 Aug 90 00:08:09 GMT
Sender: daemon@ucbvax.BERKELEY.EDU
Reply-To: wmb%MITCH.ENG.SUN.COM@SCFVM.GSFC.NASA.GOV
Organization: The Internet
```

> In a similar vein, does anyone have a forth memory allocator/freeer?

Here's a pretty good memory allocator that Don Hopkins and I wrote. I'm using it in production code, so it should be pretty solid by now. It strikes a reasonable balance between speed of allocation, speed of freeing, resistance to fragmentation, and ability to allocate arbitrary size pieces.

Enjoy, Mitch Bradley

```
\ Forth dynamic storage managment.
\ Implementation of the ANS Forth BASIS 11 memory allocation wordset.
\
\ By Don Hopkins, University of Maryland (now at Sun Microsystems)
\ Heavily modified by Mitch Bradley, Bradley Forthware
\ Public Domain.
\ Feel free to include this in any program you wish, including
\ commercial programs and systems, but please give us credit.
\
\ First fit storage allocation of blocks of varying size.
\ Blocks are prefixed with a usage flag and a length count.
\ Free blocks are collapsed downwards during free-memory and while
\ searching during allocate-memory.  Based on the algorithm described
\ in Knuth's _An_Introduction_To_Data_Structures_With_Applications_,
\ sections 5-6.2 and 5-6.3, pp. 501-511.
\
\ In the following stack diagrams, "ior" signifies a non-zero error code.
\
\ init-allocator  ( -- )
\     Initializes the allocator, with no memory.  Should be executed once,
\     before any other allocation operations are attempted.
\
\ add-memory  ( adr len -- )
\     Adds a region of memory to the allocation pool.  That memory will
\     be available for subsequent use by allocate and resize.  add-memory may
\     be executed any number of times.
\
\ allocate  ( size -- ior  |  adr 0 )
\     Tries to allocate a chunk of memory at least size bytes long.
\     Returns nonzero error code on failure, or the address of the
\     first byte of usable data and 0 on success.
\
\ free  ( adr -- ior  |  0 )
\     Frees a chunk of memory allocated by allocate or resize. adr is an
\     address previously returned by allocate or resize.  Error if adr is
\     not a valid address.
\
\ resize  ( adr1 len -- adr1 ior  |  adr2 0 )
\     Changes the size of the previously-allocated memory region
\     whose address is adr1.  len is the new size.  adr2 is the
\     address of a new region of memory of the requested size, containing
```

```
\     the same bytes as the old region.
\
\ available  ( -- size )
\     Returns the size in bytes of the largest contiguous chunk of memory
\     that can be allocated by allocate or resize .

8 constant #dalign \ Machine-dependent worst-case alignment boundary

2 base !
1110000000000111 constant *dbuf-free*
1111010101011111 constant *dbuf-used*
decimal

: field  \ name  ( offset size -- offset' )
   create over , +  does> @ +
;

struct
   /n field .dbuf-flag
   /n field .dbuf-size
aligned
   0  field .dbuf-data
   /n field .dbuf-suc
   /n field .dbuf-pred
constant dbuf-min

dbuf-min buffer: dbuf-head

: >dbuf  ( data-adr -- node )  0 .dbuf-data -  ;

: dbuf-flag!  ( flag node -- )  .dbuf-flag !   ;
: dbuf-flag@  ( node -- flag )  .dbuf-flag @   ;
: dbuf-size!  ( size node -- )  .dbuf-size !   ;
: dbuf-size@  ( node -- size )  .dbuf-size @   ;
: dbuf-suc!   ( suc node -- )   .dbuf-suc  !   ;
: dbuf-suc@   ( node -- node ) .dbuf-suc  @   ;
: dbuf-pred!  ( pred node -- )  .dbuf-pred !   ;
: dbuf-pred@  ( node -- node ) .dbuf-pred @   ;

: next-dbuf   ( node -- next-node )  dup dbuf-size@ +  ;

\ Insert new-node into doubly-linked list after old-node
: insert-after  ( new-node old-node -- )
   >r  r@ dbuf-suc@  over  dbuf-suc!   \ old's suc is now new's suc
   dup r@ dbuf-suc!                    \ new is now old's suc
   r> over dbuf-pred!                  \ old is now new's pred
   dup dbuf-suc@ dbuf-pred!            \ new is now new's suc's pred
;
: link-with-free  ( node -- )
   *dbuf-free*  over  dbuf-flag! \ Set node status to "free"
   dbuf-head insert-after  \ Insert in list after head node
;

\ Remove node from doubly-linked list
: remove-node  ( node -- )
   dup dbuf-pred@  over dbuf-suc@ dbuf-pred!
   dup dbuf-suc@   swap dbuf-pred@ dbuf-suc!
;
```

```
\ Collapse the next node into the current node

: merge-with-next  ( node -- )
   dup next-dbuf dup remove-node  ( node next-node )   \ Off of free list

   over dbuf-size@ swap dbuf-size@ +  rot dbuf-size!     \ Increase size
;

\ node is a free node.  Merge all free nodes immediately following
\ into the node.

: merge-down  ( node -- node )
   begin
      dup next-dbuf dbuf-flag@  *dbuf-free*  =
   while
      dup merge-with-next
   repeat
;

\ The following words form the interface to the memory
\ allocator.  Preceding words are implementation words
\ only and should not be used by applications.

: msize  ( adr -- count )  >dbuf dbuf-size@ >dbuf  ;

: free  ( adr -- ior  |  0 )
   >dbuf   ( node )
   dup dbuf-flag@ *dbuf-used* <>  if
      -1
   else
      merge-down link-with-free  0
   then
;

: add-memory  ( adr len -- )
   \ Align the starting address to a "worst-case" boundary.  This helps
   \ guarantee that allocated data areas will be on a "worst-case"
   \ alignment boundary.

   swap dup  #dalign round-up      ( len adr adr' )
   dup rot -                       ( len adr' diff )
   rot swap -                      ( adr' len' )

   \ Set size and flags fields for first piece

   \ Subtract off the size of one node header, because we carve out
   \ a node header from the end of the piece to use as a "stopper".
   \ That "stopper" is marked "used", and prevents merge-down from
   \ trying to merge past the end of the piece.

   >dbuf                           ( first-node first-node-size )

   \ Ensure that the piece is big enough to be useable.
   \ A piece of size dbuf-min (after having subtracted off the "stopper"
   \ header) is barely useable, because the space used by the free list
   \ links can be used as the data space.

   dup dbuf-min < abort" add-memory: piece too small"
```

```
   \ Set the size and flag for the new free piece

   *dbuf-free* 2 pick dbuf-flag!   ( first-node first-node-size )
   2dup swap dbuf-size!            ( first-node first-node-size )

   \ Create the "stopper" header

   \ XXX The stopper piece should be linked into a piece list,
   \ and the flags should be set to a different value.  The size
   \ field should indicate the total size for this piece.
   \ The piece list should be consulted when adding memory, and
   \ if there is a piece immediately following the new piece, they
   \ should be merged.

   over +                          ( first-node first-node-limit )
   *dbuf-used* swap dbuf-flag!     ( first-node )

   link-with-free
;
: allocate  ( size -- ior  |  adr 0 )
   \ Keep pieces aligned on "worst-case" hardware boundaries
   #dalign round-up                ( size' )

   .dbuf-data dbuf-min max         ( size )

   \ Search for a sufficiently-large free piece
   dbuf-head                       ( size node )
   begin                           ( size node )
      dbuf-suc@                    ( size node )
      dup dbuf-head =  if          \ Bail out if we've already been around
         2drop -1  exit            ( ior )
      then                         ( size node-successor )
      merge-down                   ( size node )
      dup dbuf-size@               ( size node dbuf-size )
      2 pick >=                    ( size node big-enough? )
   until                           ( size node )

   dup dbuf-size@ 2 pick -         ( size node left-over )
   dup dbuf-min <=  if             \ Too small to fragment?

      \ The piece is too small to split, so we just remove the whole
      \ thing from the free list.

      drop nip                     ( node )
      dup remove-node              ( node )
   else                            ( size node left-over )

      \ The piece is big enough to split up, so we make the free piece
      \ smaller and take the stuff after it as the allocated piece.

      2dup swap dbuf-size!         ( size node left-over) \ Set frag size
      +                            ( size node' )
      tuck dbuf-size!              ( node' )
   then
   *dbuf-used* over dbuf-flag!     \ Mark as used
   .dbuf-data 0                    ( adr 0 )
;

: available  ( -- size )
```

```
   0 .dbuf-data                     ( current-largest-size )

   dbuf-head                        ( size node )
   begin                            ( size node )
      dbuf-suc@  dup dbuf-head <>   ( size node more? )
   while                            \ Go once around the free list
      merge-down                    ( size node )
      dup dbuf-size@                ( size node dbuf-size )
      rot max swap                  ( size' node )
   repeat
   drop  >dbuf                      ( largest-data-size )
;

\ XXX should be smarter about extending or contracting the current piece
\ "in place"
: resize  ( adr1 len -- adr1 ior  |  adr2 0 )
   allocate  if          ( adr1 adr2 )
      2dup  over msize  over msize  min  move
      swap free  0       ( adr2 0 )
   else                  ( adr )
      -1                 ( adr1 ior )
   then
;

\ Head node has 0 size, is not free, and is initially linked to itself
: init-allocator  ( -- )
   *dbuf-used* dbuf-head dbuf-flag!
   0 dbuf-head dbuf-size! \ Must be 0 so the allocator won't find it.
   dbuf-head  dup  dbuf-suc! \ Link to self
   dbuf-head  dup  dbuf-pred!
;
```