Dynamic Memory Allocation

Klaus Schleisiek-Kern, DELTA t GmbH,
Roter Hahn 42, D-2000 Hamburg 72, FRGermany

Implementing a time-sliced multitasker in Forth reveals the inadequacy of the BLOCK concept - the validity of a block address can not be guaranteed any longer. The words ALLOCATE and FREE are defined to manage main memory which can be explicitely used to store mass storage buffers (records), data- and return-stack, string-stack and string-variables. As it turns out, an optimal algorithm for dynamic memory allocation is more compact than a clean implementation of an LRU-scheduled block buffer scheme.

Using "virtual memory" for mass storage via BLOCK is a nice way to access a disk drive if no operating system is present. The interface that reads and writes blocks of fixed size to/from mass storage is simple and can easily be implemented on any kind of hardware. Thus BLOCK is a great vehicle to use mass storage, in systems which do not (yet) have an operating system running.

Using BLOCK in a multi tasking environment is trickier. After a task switch took place, a memory buffer that held a certain BLOCK may have been used by some other task assigning a totally different block to the same memory area. Hence, the programmer must exercise care to execute BLOCK again whenever the word PAUSE had been directly or indirectly called. The FORTH-83 standard has explicitly marked these words for this purpose .

If instead an interrupt driven time sliced multi tasker is used the situation becomes even more difficult. A task switch may occur at any time. Therefore the validity of a BLOCK address can never be guaranteed unless some kind of semaphore locking scheme is incorporated in the BLOCK buffer mechanism. Although this is possible the time overhead appeared to be prohibitive.

Instead dynamic memory allocation is used to allocate disk buffers of arbitrary size which has to be taken care of by the application program. A pool of memory is set aside to be managed by the dynamic memory allocation program. Naturally this would be the memory area which traditionally has been set aside for the block buffers.

This is especially advantageous if FORTH is running under a native operating system that knows about files. Then the length of the buffer can match the record length of a certain application. The programmer is in control of what information from mass storage resides in memory at any given time. On the other hand the programmer has the burden of returning unused buffer space back to the dynamic memory manager. Otherwise the memory pool may run out of memory. This would constitute a problem.

The memory allocation strategy which I have implemented has been described by D.E.Knuth ("The Art of Computer Programming", Vol.1, Pg.442, Algorithm C). It makes efficient use of memory even if blocks of greatly varying size have to be allocated. It is reasonably fast for the sort of things that a FORTH system would use it for: File buffers, Stacks, String-variables, Matrices. I.e. "semi-static" objects which seldom have to be allocated and have a long lifetime which mostly equals the runtime of an entire application.

No provisions have been made to do garbage collection. Note: the optimal garbage collection strategy is one that never creates any garbage in the first place. If you want to implement LISP you would typically have the proposed memory allocator set aside a memory area which in turn would be managed by a dedicated allocator/deallocator/garbage-collector.

The user deals with the allocator via two words:

ALLOCATE (quantity -- address)

> allocates at least QUANTITY number of bytes in the dynamic memory pool. ADDRESS is the address of the first useable byte of contiguous main memory. The cell preceeding ADDRESS holds the actual number of bytes which may be used. If QUANTITY exceeds the size of the largest block still available in the memory pool the system aborts with the error message "out of memory".

FREE (address --)

> puts the memory block at ADDRESS back into the memory pool. If adjacent blocks are also empty they will be merged with the returned block to form an empty block of larger size. If ADDRESS is not the address of a memory block that had previously been allocated the result is unpredictable. If you are lucky the system crashes immediately.

and one word is used to set aside a portion of main memory as memory pool:

EMPTY-MEMORY (address quantity --)

> sets aside a contiguous portion of memory QUANTITY bytes long starting at ADDRESS. A list header is created and the remaining memory constitutes one large block of available memory to be ALLOCATEd.

the algorithm

All free blocks of memory are linked into a doubly-linked list. The variable ANCHOR points at some free block. When a block of memory needs to be allocated the search for a free block of sufficient size starts at ANCHOR. The first block which is large enough will be used ("first-fit"). If the block which is used is more than WASTE bytes larger then the current demand it will be split into the block to be returned and a remaining free block which will be linked into the list of available blocks. ANCHOR will be set such that it points just past the block which had been probed last. The actual block allocated is at least 2 cells larger than the number of bytes asked for so that the length of the usable block can be recorded at both ends.

Available blocks of memory have a length field on both ends of the block as well and the sign-bit is set. Hence, available blocks can be distinguished from used blocks by the state of the sign bit. When a block of memory is put back into the memory pool the neighbor towards lower memory addresses (to the "left") is checked whether it is already free.

If this is the case, the actual length of the memory block which is currently returned is added to the length of its "left" neighbor and the length field at the other end of this enlarged empty block gets marked accordingly.

If the "left" neighbor is still in use, the returned block is linked into the list of available memory blocks and the sign-bits of the length fields are set.

Then the neighbor towards higher memory addresses (to the "right") is inspected. If this block is empty as well, it is linked out of the list of available memory blocks and its length is accumulated into the current block.

The following three pages contain the source code of the above algorithm with shadow screens.

0 **9**

```
\ dynamic memory allocation                          ks 13 nov 88
         |<------------- len -------------->|
  0      |2      4                          |
 -----------------------------------------------------
 | X_len | >ptr | <ptr |    empty memory    | X_len |
 -----------------------------------------------------
          |
        Anchor

  address of >PTR is the reference address of a memory block
  which becomes the address of useable memory after allocation.

  X is MSB and set, if block is free, not set if used
  LEN is usable length in bytes
  >PTR is absolute Addr. of next empty block
  <PTR is absolute Addr. of previous empty block
```

1 **10**

```
\ dynamic memory allocation load screen              ks 13 nov 88
  Only Forth also definitions  decimal

  : cell-  ( addr1 -- addr2 )  2- ;                  some operators for transportability between
  : cell+  ( addr1 -- addr2 )  2+ ;                  16 and 32-bit systems.
  : cells  ( n1 -- n2 )        2* ;

  3 cells Constant 3cells


   2 8 thru
```

2 **11**

```
\ variables, constants  addr&len  above              ks 18 okt 88

  Variable anchor  \ points past the last referenced empty block  the list of empty blocks form a ring. ANCHOR points at the
  anchor off                                                      next block which will be looked at for allocation.

      050 Constant waste  \ don't split block if rest is below    If a block is less then WASTE bytes larger than the request,
                                                                  the remaining bytes will not be linked into the free list.
hex 08000 Constant #free  decimal                                 A mask that identifies a free memory block.
#free not Constant #max                                           The mask to mask off the free block mark.

  : addr&len  ( mem -- mem len )  dup cell- @ #max and ;          Given a block address it returns its length over the address

  : above     ( mem -- >mem )     addr&len + cell+ cell+ ;        Given a block address it returns the address of the adjacent
                                                                  block towards higher memory addresses.
```

3

```
\ use release fits?                           ks 13 nov 88

  : use    ( mem len -- )
     dup >r  swap
     2dup cell- !               \ mark lower end
     r> #max and + ! ;          \ mark upper end

  : release ( mem len -- )   #free or use ;

  : fits?  ( len -- mem / ff )  >r  \ LEN on return stack
     anchor @                   \ try at ANCHOR first
     BEGIN  addr&len  r@ u< not \ big enough?
            IF  r> drop exit  THEN  \ yes, return address
            @ dup anchor @ =    \ back at beginning of list?
     UNTIL  0= r> drop ;        \ no success, return false
```

12

A block will be marked at both ends as a used block given its block address and usable length.

Marks a block as unused.

Returns the address of a free block which is larger than LEN bytes. The search starts at ANCHOR and the first block which is large enough will be returned (first-fit). If no block in the free memory list is large enough a FALSE flag will be returned. Another possibility would be to start a garbage collection routine.

4

```
\ link @links setanchor unlink                ks 13 nov 88

  : link  ( mem >mem <mem -- )
     >r  2dup cell+ !           \          new <- above
     over !                     \          new -> above
     r> 2dup !                  \ below -> new
     swap cell+ ! ;             \ below <- new

  : @links  ( mem -- >mem <mem )   dup @   swap cell+ @ ;

  : setanchor ( mem -- mem )
     dup anchor @ = IF  dup @ anchor !  THEN ;

  : unlink  ( mem -- )   setanchor
     @links  2dup !             \ below -> above
     swap cell+ ! ;             \ below <- above
```

13

Given the address of a new block and the addresses of the previous and following blocks the new block will be linked into the doubly linked list.

Returns the addresses of the preceeding and following blocks.

Makes sure that ANCHOR does not point at the current block.

Removes the block at address MEM from the doubly linked list.

5

```
\ allocate memory                             ks 13 nov 88

  : allocate  ( len -- mem )
     3cells umax dup >r         \ never use list head
     fits? ?dup 0= Abort" memory exhausted".
     addr&len r@ -              \ #bytes block is larger
     dup waste u<               \ negligible?
     IF    drop  dup @ over unlink  \ remove from free-list
           over addr&len use    \ mark as used block
     ELSE  cell- cell-          \ remaining length
           over r@ use          \ mark allocated block
           over above           \ address of unused part
           dup rot release      \ mark as free block
           2dup swap @links link  \ link into free list
     THEN  r> drop  anchor ! ;  \ bump anchor
```

14

Given a lenght, allocate returns the address of a memory block that is at least length and at most length+waste bytes long.
The cell preceeding the block address MEM holds the byte count of the number of useable bytes of the block.
If the block that is removed from the free list is significantly (> WASTE) larger than the request, it will be split into the block to be returned and a remaining block which will be put back into the free list.

6

```
\ free memory                                  ks 13 nov 88

  : free  ( mem -- )
     addr&len                         \ #bytes to put back
     over cell- cell- @ dup 0<        \ block below empty?
     IF  #max and cell+ cell+         \ abs. length of block
         rot over - -rot +            \ merge block lengths
     ELSE  drop  over anchor @        \ at anchor,
           dup cell+ @  link          \ link into free list
     THEN
     2dup + cell+ dup @ dup 0<        \ block above empty?
     IF  #max and swap cell+  unlink  \ remove from free list
         + cell+ cell+ release exit   \ merge lengths and mark
     THEN
     2drop release ;                  \ mark as free block
```

15

MEM is the address of a block to be given back into the memory pool. MEM must be a valid memory block address, otherwise the outcome of the operation can not be predicted and a system crash is very likely.
If the adjacent block towards lower memory addresses is free already, the length of the currently released block will be merged.
Otherwise the block will be linked into the free list.

If the adjacent block towards higher memory addresses is free also it will be removed from the free list and its length will be accumulated into the block currently beeing freed.

7

```
\ initialize dynamic memory area               ks 13 nov 88

  : arguments  ( n -- )
     depth 1- > Abort" not enough parameters" ;

  : empty-memory  ( addr len -- )  2 arguments
     >r  cell+ dup anchor !         \ initialize anchor
     dup 2 cells use                \ allocate list header
     dup 2dup link                  \ initialize pointers
     dup above   swap over  dup link
     dup r> 7 cells - release       \ allocate mem-pool
     above cell- off ;              \ upper sentinel

  here 4000 allot 4000 empty-memory
```

16

Make sure enough parameters are on the stack.

Given a memory address and length this portion of memory will be initialized as a dynamic memory pool.
Free memory blocks are linked into a doubly linked list.

8

```
\ display chain of free memory blocks          ks 13 nov 88

  : end?  ( addr -- addr f )   dup anchor @ = key? or ;

  : ?memory   anchor @
     cr ." ->:"
     BEGIN  ?cr dup 6 u.r ." : "
            addr&len 4 u.r  @ end?
     UNTIL
     cr ." <-:"
     BEGIN  ?cr dup 6 u.r  ." : "
            addr&len 4 u.r  cell+ @  end?
     UNTIL  drop ;
```

17

Prints out the list of free blocks.