Region-based Memory Allocation in Forth

M. Anton Ertl*
TU Wien

Abstract

Memory management has a pervasive effect on the way we program. In region-based memory allocation, objects with roughly the same life expectancy are allocated in one region, and in the end the whole region is freed at once. This avoids the need to keep track of the individual objects for `free`. Regions are simple to implement and compatible with real-time requirements and multi-threading, and seem to be ideal for Forth, except for one thing: The region id has to be passed to the allocation word, increasing the stack load. We propose using context wrappers to avoid that problem. This even allows to use existing `allocate`-based libraries with regions, but we then have to decide what `free` and `resize` inside these libraries do.

1 Introduction

The way that memory is allocated and deallocated has far-ranging consequences on program design.

For example, consider a string concatenation word. If you can allocate memory at will, and don't have to worry about deallocation (e.g., because you work on a garbage-collected system), you might use an interface like

```
astr+ ( c-addr1 u1 c-addr2 u2
        -- c-addr3 u3 )
```

By contrast, if memory is allocated once and for all ("static allocation"), you might go for an interface like

```
bstr+ ( c-addr1 u1 c-addr2 u2 c-addr3 u3
        -- c-addr3 u4 n)
```

(inspired by the Forth-2012 word `substitute`). `Bstr+` writes the resulting string in the buffer `c-addr3 u3`, with the length of the resulting string in `u4`, and `n` indicating whether the operation was successful (had enough buffer space).

If you need to free explicitly, you can use either interface, but if you use `astr+`, you have to keep track of c-addr3 and free it when you are done.

The usage of these words varies depending on how memory is allocated. E.g., consider wanting to build a file path from a directory name `dir ( -- c-addr u)` and a file name `file ( -- c-addr u )` and then using that file path for opening a file:

```
\ astr+ with garbage collection
dir s" /" astr+ file astr+ r/o open-file throw

\ astr+ with allocate/free
dir s" /" file astr+ over >r astr+ r> free throw
over >r r/o open-file throw r> free throw

\ bstr+ with preallocated buffers:
create buf1 200 chars allot
create buf2 200 chars allot
dir s" /" buf1 200 bstr+ 0< abort" buf1 short"
file buf2 200 bstr+ 0< abort" buf2 short"
r/o open-file throw
```

Garbage collection makes such things easy, and may be the decisive feature for distinguishing high-level languages from lower-level languages, but it seems like it does not quite fit Forth: Its implementation is complex, in particular in combination with lack of type information (a fundamental property of Forth), real-time requirements (relevant in significant numbers of Forth applications), and multiprocessing (becoming more and more important with the spread of multi-core CPUs). Nevertheless, there has been a garbage collection library for Forth available since 1999[1]; however, this library does not satisfy real-time requirements and is not designed for multiprocessing.

The Forth standard supports `allocate` and `free` (and `resize`) in the memory allocation wordset since Forth-94 (*heap allocation*). Unfortunately, this interface is cumbersome and error-prone:

- If you `free` too early, the system may allocate the memory for some other use and if you then try to access the (already-freed) object, you get the wrong data or change data in the new, unrelated object (*dangling reference*).
- If you fail to keep track of all allocations, you fail to `free` some, and you get a *memory leak*.[2]

*Correspondence Address: Institut für Computersprachen, Technische Universität Wien, Argentinierstraße 8, A-1040 Wien, Austria; anton@mips.complang.tuwien.ac.at

[1] `http://www.complang.tuwien.ac.at/forth/garbage-collection.zip`

[2] Note that freeing everything just before leaving the sys-

There are various techniques to avoid these problems, but they tend to restrict the way you program, and they may cost performance; e.g., in the extreme you can make a new copy of the object every time you copy the address, and then you can be sure that you can free the object when you consume that address (because every object has only one live address) [Bak94], but all that allocating, copying, and freeing costs performance; also, this technique does not work for mutable objects.

This paper discusses region-based memory allocation, a technique in between `free` and garbage collection that might be a good fit for Forth. It describes what region-based memory allocation is (Section 2), presents Forth words for regions (Section 3), discusses how `allocate`/`free`/`resize` code can be used with regions (Section 4), outlines and implementation (Section 5) and discusses related work (Section 6).

2 Region-Based Memory Allocation

With region-based memory allocation, you can have several regions active at the same time. You allocate memory from one of these regions. When you no longer need any of the memory in a region, you free the region.

The way regions are typically used is: As application programmer you know that a bunch of things are guaranteed not to be needed beyond a certain point, so you introduce a region for these things, and allocate memory for these things from this region. In between, you can allocate things from longer-lived or shorter-lived regions. Typical examples for this kind of pattern are:

- A web server typically has a lot of things that don't survive the HTTP request. These things could be allocated in a region that is freed when servicing the request is completed.
- A compiler could have regions for the basic block (straight-line code sequences), and the definition. As soon as it is done with one basic block, it frees the basic block region and starts a new basic block region for the next basic block. Likewise for definitions.
- A text formatting program could have regions for a line, a paragraph, a page, a section, and the whole document.

Regions give programmers a wide range of control over memory management. E.g., you could start out with few regions (e.g., in the compiler only have regions for definitions); when you notice that this consumes more memory than you want, you can introduce additional regions for more fine-grained control (but with the potential for more bugs).

Regions are relatively easy to implement (about the same difficulty as `allocate`/`free`), even in the presence of real-time requirements and multiprocessing. So they appear to be a good fit for Forth. Why have they not caught on?

3 Forth interface for regions

A straightforward region interface works with region IDs passed on the stack:

```
new-region ( -- region-id )
region-alloc ( usize region-id -- addr )
free-region ( region-id -- )
```

The disadvantage of this kind of interface is that it requires passing the region-id around. E.g., for our string concatenation example, we would have a word

```
cstr+ ( c-addr1 u1 c-addr2 u2 region-id
        -- c-addr3 u3 )
```

The region-id would have to be passed around on the stack inside `cstr+`, and we would have to pass the region-id to `cstr+`. For our file path example this could look as follows:

```
new-region >r
dir s" /" r@ cstr+ file r@ cstr+
r/o open-file throw
r> free-region
```

This works passably in this case, but we consumed the top-of-return-stack for the region-id, and cannot use it for something else anymore. In any case, this kind of region interface increases the stack load by one item.

This has deterred me from using regions for a long time, but recently I have thought about how to use stack load reduction techniques [Ert11] to avoid this problem. I settled for using context wrappers, because this allows writing general-purpose words. `Region-alloc` is split into:

```
ralloc ( usize -- addr )
with-region ( ... region-id xt -- ... )
\ xt: ( ... -- ... )
```

So you pass the region-id to `with-region`, which executes the xt, and while executing the xt, every `ralloc` allocates from region-id (unless it is executed in a nested `with-region` context).

Let's look at our string concatenation example again. We can now use the `astr+` interface instead fo `cstr+`:

tem is counterproductive; it may page in stuff that would just be freed (without paging) by the operating system as part of terminating the process.

```
new-region dup
[: dir s" /" astr+ file astr+
   r/o open-file throw ;] with-region
free-region
```

This example uses the syntax `[: ... ;]` for nestable unnamed definitions (quotations). The example is not shorter than the `cstr+` one, but the return stack is now free for other uses (within the quotation).

But there is still a stack item passed from `new-region` to `free-region`. We can also have a wrapper that replaces these two words:

```
do-region ( ... xt -- ... )
\ xt stack effect: ( ... region-id -- ... )
```

With that, our example looks as follows:

```
[: [: dir s" /" astr+ file astr+
      r/o open-file throw
   ;] with-region
;] do-region
```

For cases like this example where `do-region` and `with-region` work together, we can also have

```
do-with-region ( ... -- ... )
\ xt stack effect: ( ... -- ... )
```

which combines the effects, resulting in:

```
[: dir s" /" astr+ file astr+
   r/o open-file throw ;] do-with-region
```

4 Allocate/free/resize

With the region passed implicitly, we can use an interface that is compatible to the standard word

```
allocate ( usize -- addr ior )
```

instead of `ralloc`. Indeed, we can even redefine `allocate` to allocate from the current region when called inside a `with-region` context. This allows to use words or libraries written for the standard memory-allocation wordset with regions.

To make this idea work, we also need to determine what `free` and `resize` should do when called inside a `with-region` context.

For `free` this is relatively straightforward: if the memory has been allocted from a region, `free` should not free anything (the memory will be freed when the region is freed); if the memory has been allocated from the heap, then `free` should perform the standard `free`.

`Resize` is more complicated. One can see it as allocating memory from the current region, and freeing the original memory as described above. However, that would not always reflect the intent of the programmer who wrote the `resize`, and may lead to too-early freeing.

So how is `resize` used in practice? In my experience `resize` is used in two ways:

- To simulate statically allocated buffers of unlimited size. The program first `allocate`s a small buffer (or stores 0 as buffer address), and grows the buffer with `resize` when necessary. These buffers are never `free`d.
- For temporary growing structures. These structures are `free`d when the program no longer needs them.

Given that, one approach for dealing with `resize` is to always treat it as working on the heap. If the memory was first allocated from a region, the `resize` should be treated as allocating from the heap. People who want to write code for regions should not use `resize`.

One problem with these ideas is that it sometimes requires determining whether a piece of memory was allocated from the heap or from a region. Determining this can require quite a bit of code and can be slow (depending on the implementation of regions and the heap).

The following assumptions would get rid of this need:

- `Resize` only gets 0 or previously `resize`d memory as a-addr1 parameter. With this assumption `resize` does not need to see if the memory was allocated from a region (it wasn't). Unfortunately, the standard does not specify that `resize` works for a-addr1=0 (Gforth does), so this assumption will not hold for standard programs that use `resize`.

  An alternative, less restrictive assumption is that the `resize`d memory was `allocate`d from the heap, but that would restrict the usage of `with-region` in combination with code that uses `resize` for temporary growing structures. To avoid programs that don't get this right, it would be useful to check this assumption, but that again requires determining whether memory was allocated from the heap or from a region.

  If this assumption is made, but does not hold (i.e., region-allocated memory is `resize`d), the result is unpredictable and depends on the heap implementation.

  The other alternative is to assume that the memory is either from a region or previously resized. Then, if it is not previously resized, we just heap-allocate new memory, copy the old memory there, and do not free the old memory. If the old memory was actually heap-allocated, this will lead to a memory leak.

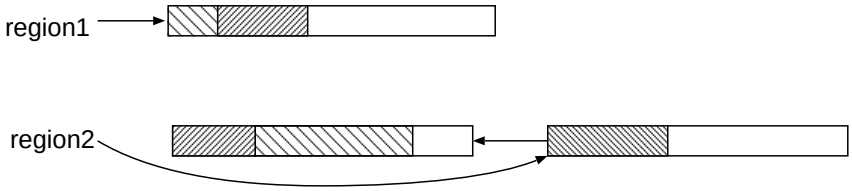

Figure 1: Implementation based on `allocate`

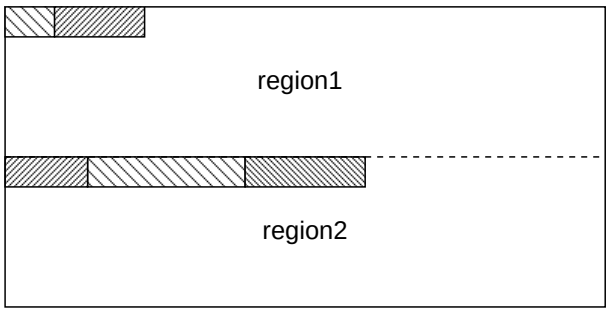

Figure 2: Implementation based on one big memory block

- `Free` within a region only refers to region-allocated memory, except possibly `resized` memory. With this assumption, `free` needs to check only if memory is `resized`, which is cheaper to check. Ideally `resized` memory is always freed with a separate word, then we can do with a placebo `free` inside a region. If this assumption is made, but does not hold (i.e., heap-allocated memory should be `freed` in region context), there will be a memory leak.

It is unclear which of the various options in this design space is best. So it is probably best to use the simplest option at first, build in checking to make users aware of the restrictions, and ask users for feedback.

5 Implementation

This section sketches two implememtation approaches.

5.1 Based on `allocate`

Each region is represented by a linked list of blocks. Each block has a standard size (e.g., 16KB) and is `allocated`. Within each block, there is a pointer to the first free byte, and a new allocation in the region is made there. If the rest of the block is too small for the allocation, a new block is started (see Fig. 1). If an allocation is bigger than the standard block, it gets its own private block of the appropriate size.

When a region is freed, the linked list is traversed and all the blocks in the linked list are `freed`. For real-time requirements, one could arrange to delay the freeing, such that only one block is freed per region allocation.

For checking whether an address is allocated with `resize`, one could have a simple array of resize addresses. If there are only few resize addresses at the same time, this is sufficient. A more scalable data structure (inspired by a sparse set representation [BT93]) would have an extra cell before the resized memory that points to the array; if this address points within the bounds of the array, and the place where it points to points back to the address we are looking at, the address has actually been allocated with `resize`.

For checking whether an address is allocated in a region or on the heap, we would have to walk all the blocks of all the heaps, and check whether the address is contained there.

The benefits of this kind of implementation over one that uses one `allocate` per `region-alloc` and links all the allocations together is less memory overhead for links, and less time overhead in allocation and deallocation.

5.2 Based on one big memory block

In an embeded system with full control over memory we may prefer to reserve one big block of memory for regions. Similarly, if we are working on a decent virtual memory system, we could mmap a big chunk of address space for regions (say, as big as the physical memory of the machine).

This implementation is based on buddy memory allocation. The first region starts out at the bottom of the big block. When starting another region, the block is divided into two parts (see Fig. 2). If

the part of one region runs out of space, one can split the part of a region with more free space, and continue there.

When freeing a region, all the parts it has are freed, possibly regrowing parts of other regions.

Checking for `resize` addresses is the same as for the other implementation.

Checking whether an address is allocated in a region or on the heap is very easy: If the address is within the big block, it is in a region.

Overall this implementation approach is similar to the other one, but you implement the base memory allocator yourself (as buddy allocator) instead of using the system's `allocate`. The benefits are that you can use your knowledge of the base allocator's implementation to simplify some of the operations of the region allocator (e.g., checking whether something is in a region).

6 Related work

Region-based memory allocation is an old idea, that has appeared under different names: regions [GA98], arenas [Han90], pools (Apache), memory contexts (PostgreSQL), obstacks (glibc). "Region" is the name used in most recent papers and in Wikipedia[3].

Glibc's obstacks extend the usual capabilities of regions by allowing to grow allocations, and deallocate from an obstack in a stack-based way, i.e., a very dictionary-like behaviour, except that you can have several obstacks, and a growable object is not addressable while it is still growable.

The regions implementation based on `allocate` is the same as that described by Hanson [Han90], and as described in the obstacks documentation of glibc.

Gay and Aiken [GA98] evaluate regions empirically, and find that regions are either best or close to the best alternative in both run-time and memory consumption. They also propose and evaluate a safe version of this technique, based on reference counting (references into a whole region).

Because regions and their implementation are so simple, there is little academic literature on them themselves, but rather on more complex ideas like region inference, where the compiler tries to determine regions for allocations automatically.

Context wrappers are one of the techniques for reducing the stack load [Ert11]. They were inspired by Jenny Brien, who proposed a wrapper for dealing with the input stream on comp.lang.forth `<8s7mkl$4ql$1@news6.svr.pol.co.uk>`.

[3] `http://en.wikipedia.org/wiki/Region-based_memory_management`

7 Conclusion

Region-based memory allocation offers a more convenient memory allocation model than `allocate`/`free`, while avoiding the problems of garbage collection: regions are much simpler to implement, especially in combination with multi-threading and real-time requirements.

So regions seem to be a good fit for Forth. However, they have not caught on yet, because they require passing the region id around, thus increasing the load on the stack. By using context-wrappers we can reduce this stack burden.

This opens up the possibility to use existing, `allocate`-using code with regions, often avoiding the need to keep track of each piece of allocated memory for `free`. But one then has to do something about the `free`s and `resize`s in this code. We have discussed this issue here, but are not sure what the best approach is.

References

[Bak94] Henry Baker. Linear logic and permutation stacks — the Forth shall be first. *ACM Computer Architecture News*, 22(1):34–43, March 1994.

[BT93] Preston Briggs and Linda Torczon. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems*, 2(1–4):59–69, 1993.

[Ert11] M. Anton Ertl. Ways to reduce the stack depth. In *27th EuroForth Conference*, pages 36–41, 2011.

[GA98] David Gay and Alex Aiken. Memory management with explicit regions. In *SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 313–323, 1998.

[Han90] David R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice and Experience*, 20(1):5–12, January 1990.