memory management

table of content

1 memory allocation

memory allocation can be an expensive operation. in many cases, it is better to block/cache/delay calls to OS memory management functions. in general memory management is a way to cache multiple requests for memory allocation. memory allocation divides into two phases: pre-allocation and request processing. normally, memory pre-allocation lowers memory request frequency by utilizing unused application memory.

^

1.1 metadata-free allocation

allocation without metadata being written to the data memory pool except for some static data structures or pointers.

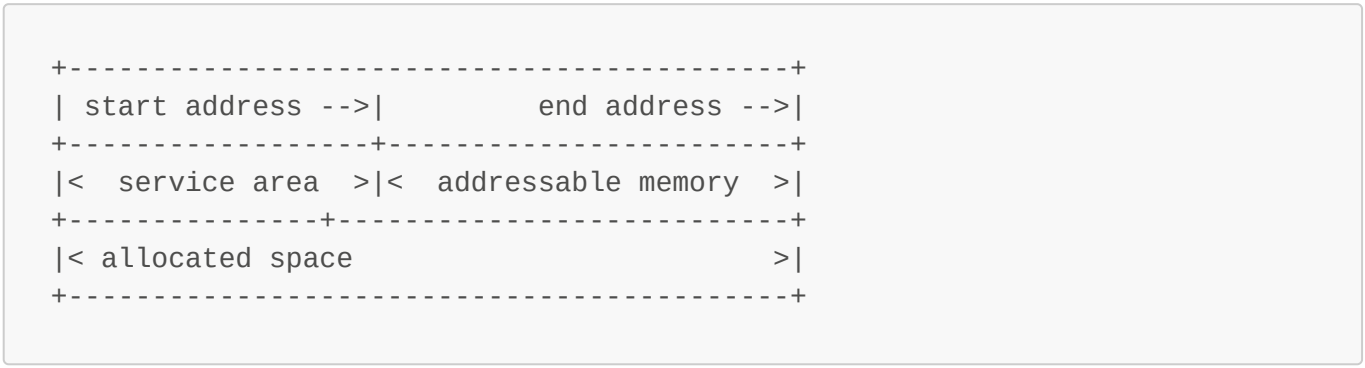
^

```
+-----------------------------------------+
|<-- start address          end address -->|
+------------------+----------------------+
|<  service                         area  >|
+-----------------------------------------+
|<  addressable memory                    >|
+-----------------------------------------+
|<  allocated space                       >|
+-----------------------------------------+
```

1.2 in-place metadata allocation

allocation where sensitive allocation metadata is placed in the same memory pool where allocated data resides.

^



1.2.1 reverse order metadata allocation

allocation where the allocator calculates the position of the metadata token and places the metadata at the end of this region, returning a pointer to the start of the region, that is why the currently allocated region being written by the user program is placed within the allocated region.

^

1.2.2 direct order metadata allocation

the same as above, except that metadata, is written first, protecting the current metadata block from casual overwrites. it is slightly better in terms of safety but will also have the same caveats in case of buffer underflow, allowing corruption of control data.

^

1.2 detached metadata allocation

allocation which completely separates data allocation pools as sensitive allocation metadata for manipulation allocations. this is the safest approach to the allocation of data.

it is similar to pp.1.1 except that pp.1.1 completely omits metadata semantics with exception of fixed-size generic types or static data structures.

^

1.3 allocation pools

allocator can use a single memory pool for service any allocation requests by using single or multiple memory pools and buffers for storing metadata information. it is useful in case one buffer has some invalid data or somewhat worse scenario metadata, then another pool can be used to take advantage of several allocation pools.

^

1.4 buffered allocation

allocation can take advantage of using several linear memory buffers to store metadata information. for example, to keep track list of items to de-allocated memory and take pointers from that list one can do an implementation allowing non-linear allocation/de-allocation.

^

2 pointer metadata

allocators can handle, operate and save additional metadata information related to memory pointers and memory allocation. probably it would be very helpful to return additional metadata info instead of naked pointers to allow monitor of the memory allocation boundaries and keeping track of the size of the region addressed by the pointer.

^

2.1 language support for metadata

C language did not include pointer metadata info in C language standards (C89) itself allowing the different classes of errors like pointer arithmetic, buffer overflow, and buffer underflow errors.

^

```
struct ptr_metadata {
    void* ptr;
    u64 size;
}
```

2.2 memory allocation blocks

allocation blocks can be used as building blocks for generic programming but the size of that blocks can vary as well

^

2.2.1 fixed memory allocation blocks

allocator uses the constant memory allocation blocks with one or more fixed-size types. possibly, it is practical to use a different type of allocation sizes complying with each or some of the rest sizes, in arithmetical or geometrical progression, i.e. 1,2,3,4,5,6,7,8 or 1,2,3,5,8,13,21,34 or 1,2,4,8,16,32,64,128

^

2.2.2 variable memory allocation blocks

allocator can be used to allocate blocks of memory with different sizes. in general, if the allocator uses memory blocks of different sizes, it actually can form the resulting memory block by combining several allocation blocks from different fixed-size allocation buffers. on the other hand, we can use single

subsequent allocation blocks to cover the required allocation size and overlap the requirements with possibly several bytes within the remaining bytes block still not allocated.

^