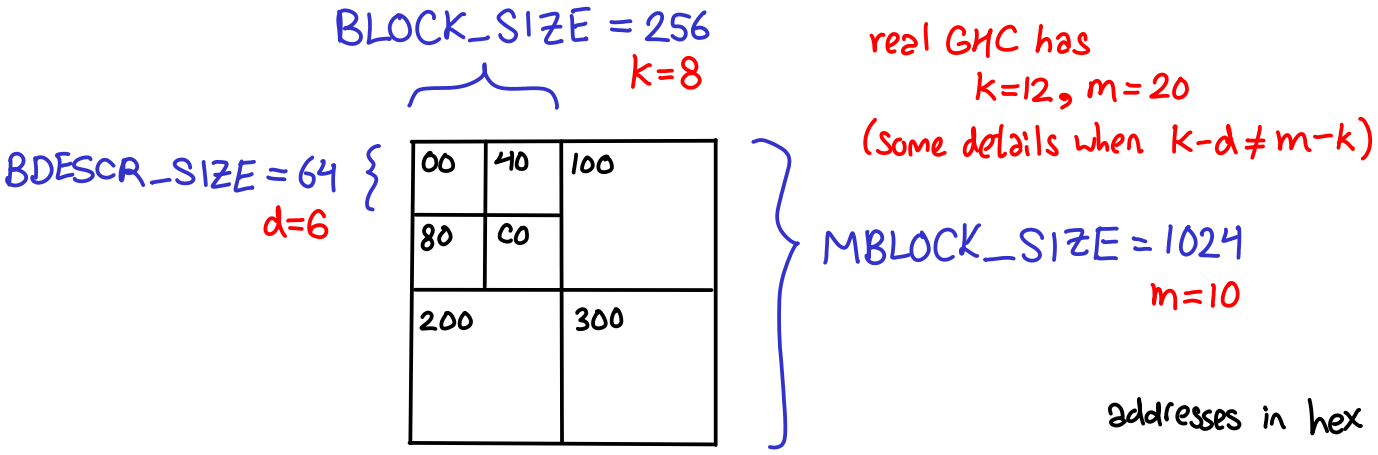
THE GHC BLOCK ALLOCATOR ON A 64-BIT MACHINE (d=6)

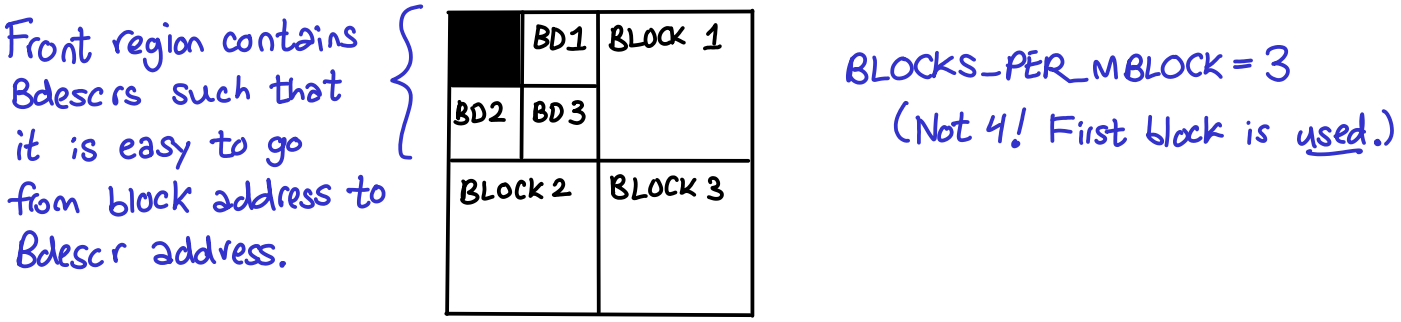
— Edward Z. Yang

The block allocator allows us to manage our heap (and other allocations) with multiple blocks, rather than a single contiguous region. This scheme was presented in the paper "Parallel generational-copying garbage collection with a block-structured heap" and there is also some good commentary at http://ghc.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/BlockAlloc

In this document, I want to visualize block allocation using small choices for block size and megablock size.



We start by looking at the memory layout of a single megablock. We've laid out the memory like a quadtree, so that given any block, the order of memory can be read out by dividing it into four sub-blocks and recursively reading the sub-blocks in the usual order (each shown block has been annotated with a hexadecimal address for your convenience). The smallest "block" is 64 bytes large (appropriate for a 64-bit machine), and represents a block descriptor (bdescr); the medium-sized block is 256 bytes large (enough to fit four block descriptors), and the big megablock (mblock) is 1024 bytes large (enough to fit four blocks). Actual GHC does not require that the number of block descriptors that fit in a block be the same as the number of blocks that fit in a megablock, but we will skip those details: we will say (inaccurately) that the block descriptors are placed in the first block. Blocks are then chained together to make up bigger regions of memory, e.g. the heap.

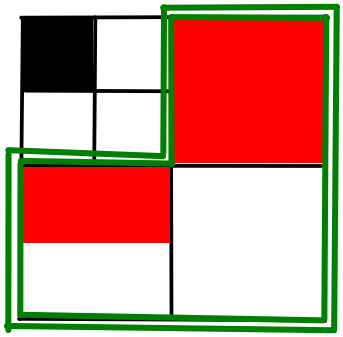


The basic idea is to allocate a megablock, and use the first block to record the block descriptors for the other blocks; it is a recursive scheme, very similar to what is used for page tables in CPUs. The thing that makes this scheme work is that all of the megablocks are aligned at megablock boundaries, so using some bit twiddling we can calculate a pointer to the block descriptor for a block using a pointer to the block.

A block descriptor contains some important metadata: a single block will record a pointer to the start of the block (technically not necessary, but we usually have space to spare and it saves some computation), a pointer to the first free location in the block, and some information for the garbage collector (e.g. what generation the block belongs to, and what generation its contents will be evacuated to.)

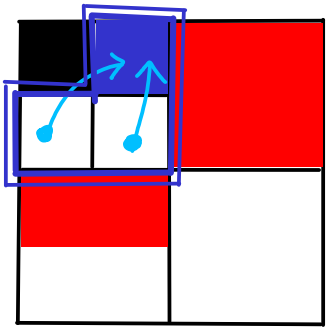
HANDLING LARGE OBJECTS

The reason why we did not put block descriptors in a more conventional location (e.g. at the head of the block) has to do with large objects.



There are two classes of large objects we have to be worried about. The first type is shown here in red, an object that is larger than a block, but smaller than a megablock. (Technically, smaller than a megablock minus one block.) Clearly, if we had placed the block descriptors at the head of blocks, there would be no way to get this contiguous space. With the block descriptors out of the way in the first block, it is easy to take multiple blocks and allocate them together as a block group (outlined in green). A block group is different from a block chain: a block chain is usually not contiguous and is composed of blocks and block groups.

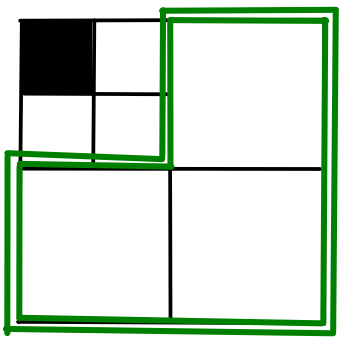
block groups ≠ block chains



The metadata of a block group looks different than the metadata for individually allocated blocks: the head of the block group (shown in solid blue) looks normal, but has a blocks field which indicates how many blocks are part of the group. Members of the block group (outlined in blue), however, have their free pointers set to zero and a link back to the head of the block group. Generally, the rest of their metadata will be setup, because objects may live anywhere inside a block group and a bdescr calculation could produce an inner block descriptor, rather than the head of the block group.

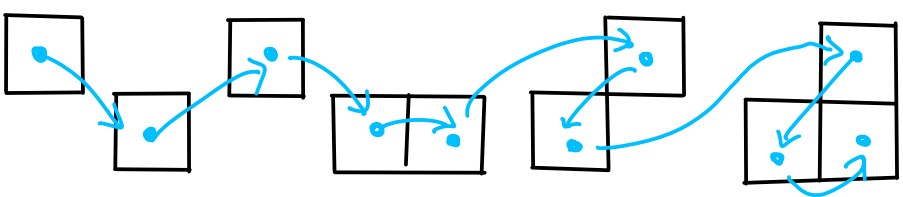
The presence of block groups means that fragmentation can be a problem. Notice, however, that a block group only involves a constant amount of metadata stored in the head of the block group. So it is not too difficult to imagine that when block groups are freed, we can look at the blocks before and after it, and see if those blocks are also free. If they are free, the free blocks can be combined into one large free block. Managing all of the invariants here is a bit tricky, see the "Free lists" note in rts/sm/BlockAlloc.c for the gory details, but the end result is that without compacting, we always know what the largest block groups available are.

HANDLING THE NURSERY



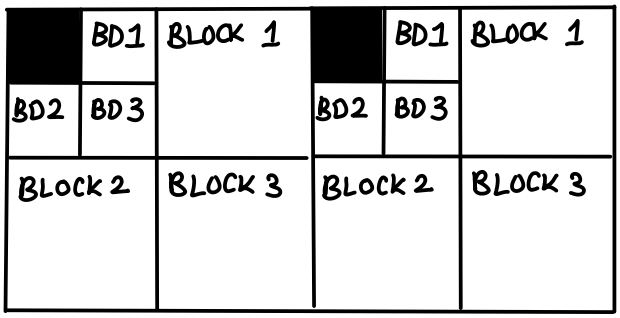
While a block group is generally only ever requested when we have a large object that must have contiguous memory, there is one special exception: allocation of nurseries. If memory is accessed serially, CPUs can do automatic prefetching across cache lines, making for something of a ~0.5% optimization is speed, so we would like the nursery to be a contiguous region. Thus, the nursery requests as large a contiguous block group as possible (blocks per mblock). There are, however, two important ancillary considerations.

First, it is not actually desirable for the nursery to be a single large block, for the purposes of parallel garbage collection. A single large block cannot load balanced over multiple processors, whereas a collection of small blocks can be. Thus, after receiving a block group, the nursery then chops the block group into individual blocks, which just happen to be contiguous. Second, requests for large blocks can result in bad fragmentation, since the bigger the request, the harder it is to fulfill. So while it is nice to have contiguous memory, it is more important to make sure that small block groups get used up. So in practice, the nursery will be a chain of blocks, which happen to be somewhat contiguous.

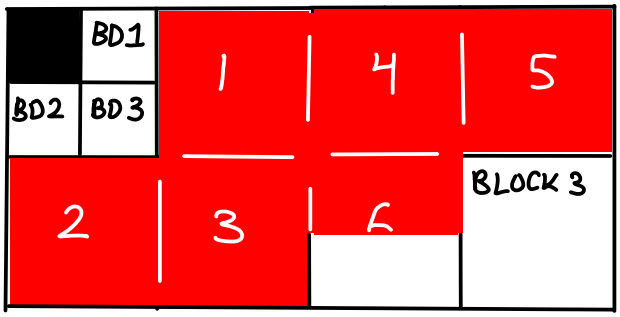


HANDLING REALLY LARGE OBJECTS

Sometimes, we need to allocate a region of memory which will end up using space larger than a megablock. This is a bit of a problem, because megablocks are how memory at large is split up:



And so an object which spills over the megablock boundary will overwrite the block descriptor table for the next megablock.



We call a bunch of megablocks which are put together contiguously in this manner a megagroup. The fact that block descriptor table for the second (and later) megablocks is overwritten leads to some interesting details.

First, the space calculation for how many blocks a group of megagroup contains is not number megablocks times the constant BLOCKS_PER_MBLOCK; recall blocks per mblock does not count the block descriptor block as part of its count; however, the second and further megablocks also get to use the block descriptor block as space. This should be pretty clear from the diagram, but it explains some of the funny arithmetic you have to do when performing these calculations. (Exercise for the reader: explain why n_alloc_blocks "doesn't count the extra blocks we get in a megablock group." Hint: it's a big hack.)

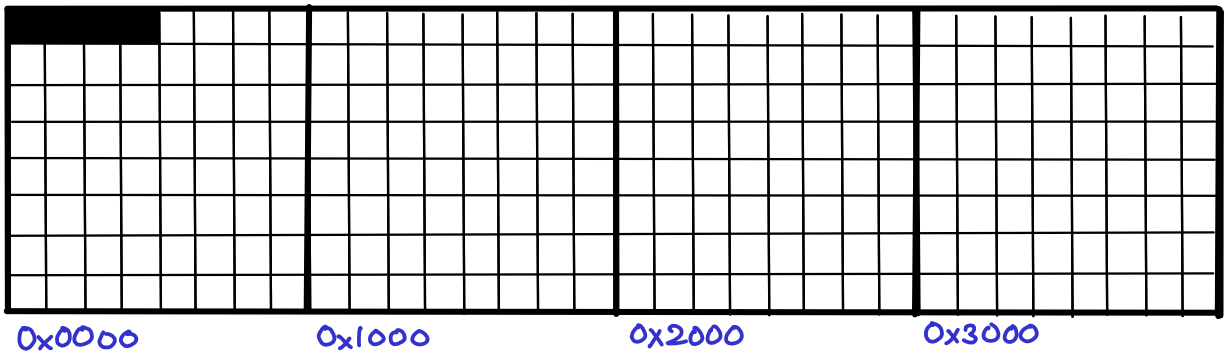
Second, we cannot use use any of the leftover space in the megagroup to store extra objects (as we could when we just had a group), because the bdescr computation will fail. That means we cannot put any megagroups in the nursery, since our bump allocator will blithely blaze past the first megagroup. Compiled code that know that it may allocate extremely large objects must do this allocation out-of-line, and the heap-overflow code in rts/Schedule.c will complain loudly if you try to allocate an object too large from the nursery.

Finally, it means we have to do a different defragmentation strategy, as the last megablock of a megagroup will not, in general, have any sort of block descriptor or other metadata telling us where the front of the megagroup is (after all, it all got overwritten). So we do something dumb which works, which is store store megagroups in ascending order of address, and allocation is done by walking the whole list. As megablocks are usually a megabyte large, allocations larger than this should be relatively rare.

THE REAL PARAMETERS

Recall that for 64-bit GHC, we have megablocks 1 MB in size (m = 20), with blocks of 4k in size (k = 12) and block descriptors of 64 bytes (d = 6).

Thus, there are 256 blocks in a megablock, thus requiring 256 block descriptors. These block descriptors take up four blocks worth of space. The first usable block is thus the fifth block. (I am no longer using a quad-tree style presentation for these diagrams.)



This results in quite a distinctive pattern when looking at GHC from the debugger: blocks are aligned by 0x1000, and if the fourth digit of the hex is zero, one, two or three, it must be a block descriptor.