Writing a Memory Allocator

This is the 6th lecture from the Garbage Collection Algorithms class, devoted to the automatic memory management.

Before discussing algorithms of *collecting* the garbage, we need to see how these objects (which eventually become a *garbage*) are *allocated* on the heap. In today's lecture we consider mechanisms of *memory allocation*.

Note: see also related lectures on **Writing a Pool Allocator**, and **Writing a Mark-Sweep Garbage Collector**.

Audience: advanced engineers

This is a lab session, where we're going to implement a memory allocator, similar to the one used in malloc function. In addition, we discuss the theory behind the allocators, talking about *sequential* (aka *"bump allocators"*), and the *free-list* allocators.

Prerequisites for this lab session are the:

Video lecture

Before going to the lab session, you can address corresponding *video lecture* on Memory Allocators, which is available at:

Lecture 6/16: Allocators: Free-list vs. Sequential

Mutator, Allocator, Collector

From the previous lectures we know that a garbage-collected program is manipulated by the three main modules known as *Mutator, Allocator,* and *Collector*.

Mutator is our *user-program*, where we create objects for own purposes. All the other modules should *respect* the Mutator's view on the object graph. For example, under no circumstances a **Collector** can reclaim an *alive* object.

Mutator however doesn't allocate objects by itself. Instead it delegates this generic task to the **Allocator** module – and this is exactly the topic of our discussion today.

Here is an overview picture of a GC'ed program:

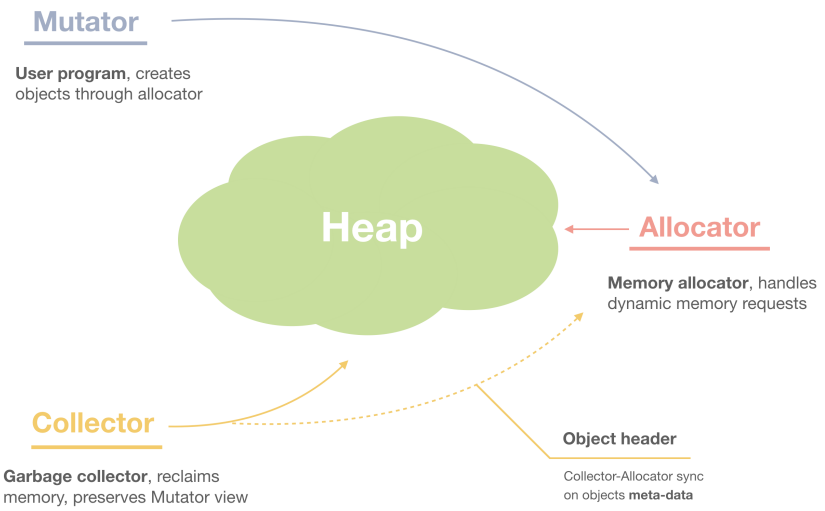

Figure 1. Mutator, Allocator, Collector

Let’s dive into the details of implementing a memory allocator.

Memory block

Usually in *higher-level* programming languages we deal with *objects*, which have a structure, fields, methods, etc:

```
1const rect = new Rectangle({width: 10, height:
 20});
```

From a memory allocator perspective though, which works at *lower-level*, an object is represented as just a *memory block*. All is known is that this block is of certain *size*, and its contents, being opaque, treated as *raw sequence of bytes*. At runtime this memory block can be *casted* to a needed type, and its logical layout may differ based on this casting.

Recall from the lecture devoted to the *Object header*, that memory allocation is always accompanied by the *memory alignment*, and the *object header*. The header stores *meta-information* related to each object, and which servers the allocator’s, and collector’s purposes.

Our memory block will combine the *object header*, and the actual *payload pointer*, which points to the first word of the user data. This pointer is returned to the user on allocation request:

```
1 /**
2  * Machine word size. Depending on the
3 architecture,
4  * can be 4 or 8 bytes.
5  */
6 using word_t = intptr_t;
7
8 /**
9  * Allocated block of memory. Contains the object
10header structure,
11 * and the actual payload pointer.
12 */
13struct Block {
14
15  // -----------------------------------
16  // 1. Object header
17
18  /**
19   * Block size.
20   */
21  size_t size;
22
23  /**
24   * Whether this block is currently used.
25   */
26  bool used;
27
28  /**
29   * Next block in the list.
30   */
31  Block *next;
32
33  // -----------------------------------
34  // 2. User data
35
36  /**
```

```
37   * Payload pointer.
38   */
39  word_t data[1];


  };
```

As you can see, the header tracks the size of an object, and whether this block is *currently* allocated — the used flag. On allocation it's set to true, and on the free operation it's reset back to false, so can be *reused* in the future requests. In addition, the next field points to the *next block* in the linked list of all available blocks.

The data field points to the *first word* of the returning to a user value.

Here is a picture of how the blocks look in memory:

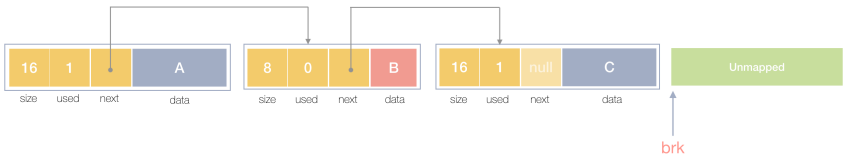

Figure 2. Memory blocks

The objects A, and C are in use, and the block B is currently not used.

Since this is a linked list, we'll be tracking the *start* and the *end* (the *top*) of the heap:

```
1/**
2 * Heap start. Initialized on first allocation.
3 */
4static Block *heapStart = nullptr;
5
6/**
7 * Current top. Updated on each allocation.
8 */
9static auto top = heapStart;
```

Shall I use C++ for implementation?

Not really! We use C++ in this specific implementation only because it's convenient for manipulating the raw memory and pointers. In the class however we study *abstract GC algorithms*, which means you can implement them in *any* language. For example, you can allocate a *virtual heap* in JavaScript using ArrayBuffer, or similarly bytearray in Python, Rust, etc.

Allocator interface

As mentioned, when allocating a memory, we don't make any commitment about the logical layout of the objects, and instead work with the *size* of the block.

Mimicking the malloc function, we have the following interface (except we're using typed word_t* instead of void* for the return type):

```
1/**
2 * Allocates a block of memory of (at least)
3`size` bytes.
4 */
5word_t *alloc(size_t size) {
6  ...
 }
```

Why is it *"at least"* of size bytes? Because of the *padding* or *alignment*, which we should discuss next.

Memory alignment

For faster access, a memory block should be *aligned*, and usually by the size of the *machine word*.

Here is the picture how an aligned block with an object header looks like:

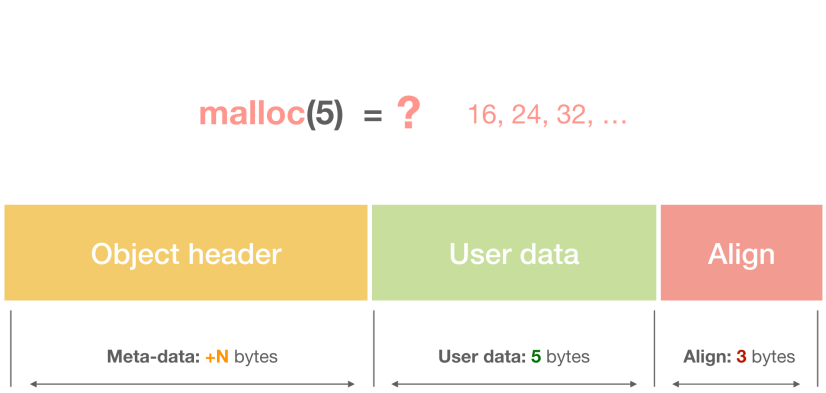

Figure 3. Aligned block with a header

Let’s define the aligning function, which looks a bit cryptic, but does the job well:

```
1/**
2 * Aligns the size by the machine word.
3 */
4inline size_t align(size_t n) {
5  return (n + sizeof(word_t) - 1) & ~
6(sizeof(word_t) - 1);
 }
```

What this means, is if a user requests to allocate, say, *6 bytes*, we actually allocate *8 bytes*. Allocating 4 bytes can result either to 4 bytes — on 32-bit architecture, or to 8 bytes — on x64 machine.

Let’s do some tests:

```
1 // Assuming x64 architecture:
2
3 align(3);  //  8
4 align(8);  //  8
5 align(12); // 16
6 align(16); // 16
7 ...
8
9 // Assuming 32-bit architecture:
10align(3);  //  4
11align(8);  //  8
12align(12); // 12
13align(16); // 16
14...
```

So this is the first step we're going to do on allocation request:

```
1word_t *alloc(size_t size) {
2  size = align(size);
3  ...
4}
```

OK, so far so good. Now we need to recall the topics of virtual memory and *memory mapping*, and see what allocation actually means from the operating system perspective.

Memory mapping

From the Lecture 4 about *virtual memory* of a process, we remember that *"to **allocate** a memory"* from OS, means *"to **map** more memory"* for this process.

Let's recall the memory layout again, and see where the heap region resides.

Figure 4. Memory Layout

As we can see, the heap grows *upwards*, towards the *higher addresses*. And the area in *between* the stack and the heap is the *unmapped* area. Mapping is controlled by the *position* of the **program break (brk)** pointer.

There are several system calls for memory mapping: brk, sbrk, and mmap. Production allocators usually use combination of those, however here for simplicity we'll be using only the sbrk call.

Having current *top of the heap*, the sbrk function *increases ("bumps")* the value of the program break on the passed amount of bytes.

Here is the procedure for requesting memory from OS:

```
1 #include <unistd.h>   // for sbrk
2 ...
3
4 /**
5  * Returns total allocation size, reserving in
6 addition the space for
7  * the Block structure (object header + first
8 data word).
9  *
10 * Since the `word_t data[1]` already allocates
11one word inside the Block
12 * structure, we decrease it from the size
13request: if a user allocates
14 * only one word, it's fully in the Block struct.
15 */
16inline size_t allocSize(size_t size) {
17  return size + sizeof(Block) -
18sizeof(std::declval<Block>().data);
19}
20
21/**
22 * Requests (maps) memory from OS.
23 */
24Block *requestFromOS(size_t size) {
25  // Current heap break.
26  auto block = (Block *)sbrk(0);
27// (1)
28
29  // OOM.
    if (sbrk(allocSize(size)) == (void *)-1) {
  // (2)
```

```
      return nullptr;
    }

    return block;
  }
```

By calling the sbrk(0) in (1), we obtain the pointer to the *current* heap break — this is the *beginning* position of the newly allocated block.

Next in (2) we call sbrk again, but this time already passing the amount of bytes on which we should *increase* the break position. If this call results to (void *)-1, we signal about *OOM (out of memory)*, returning the nullptr. Otherwise we return the obtained in (1) address of the allocated block.

To reiterate the comment on the allocSize function: in addition to the *actual* requested size, we should add the size of the Block structure which stores the object header. However, since the *first word* of the user data is already *automatically* reserved in the data field of the Block, we decrease it.

We'll be calling requestFromOS *only* when no free blocks available in our linked list of blocks. Otherwise, we'll be reusing a free block.

Note: on Mac OS sbrk is deprecated and is currently *emulated* via mmap, using a pre-allocated arena of memory under the hood.

To use sbrk with clang on Mac OS without warnings, add the:

```
1#pragma clang diagnostic push
2#pragma clang diagnostic ignored "-Wdeprecated-
 declarations"
```

As one of the assignments below you'll be suggested to implement a *custom* version of sbrk, and also on top of the mmap call.

OK, now we can request the memory from OS:

```
1 /**
```

```
2  * Allocates a block of memory of (at least)
3 `size` bytes.
4  */
5 word_t *alloc(size_t size) {
6   size = align(size);
7
8   auto block = requestFromOS(size);
9
10  block->size = size;
11  block->used = true;
12
13  // Init heap.
14  if (heapStart == nullptr) {
15    heapStart = block;
16  }
17
18  // Chain the blocks.
19  if (top != nullptr) {
20    top->next = block;
21  }
22
23  top = block;
24
25  // User payload:
26  return block->data;
  }
```

Let’s add a helper function to obtain a header from a user pointer:

```
1/**
2 * Returns the object header.
3 */
4Block *getHeader(word_t *data) {
5  return (Block *)((char *)data +
6sizeof(std::declval<Block>().data) -
7                   sizeof(Block));
 }
```

And experiment with the allocation:

```
1 #include <assert.h>
2 ...
```

```
3
4 int main(int argc, char const *argv[]) {
5
6   // ------------------------------------
7   // Test case 1: Alignment
8   //
9   // A request for 3 bytes is aligned to 8.
10  //
11
12  auto p1 = alloc(3);                       //
13(1)
14  auto p1b = getHeader(p1);
15  assert(p1b->size == sizeof(word_t));
16
17  // ------------------------------------
18  // Test case 2: Exact amount of aligned bytes
19  //
20
21  auto p2 = alloc(8);                       //
22(2)
23  auto p2b = getHeader(p2);
24  assert(p2b->size == 8);
25
26  ...
27
28  puts("\nAll assertions passed!\n");


  }
```

In (1) the allocation size is *aligned* to the size of the word, i.e. 8 on x64. In (2) it's already aligned, so we get 8 too.

You can compile it using any C++ complier, for example clang:

```
1# Compile:
2
3clang++ alloc.cpp -o alloc
4
5# Execute:
6
```

```
7./alloc
```

As a result of this execution you should see the:

```
1All assertions passed!
```

OK, looks good. Yet however we have one *“slight”* (read: *big*) problem in this *naïve* allocator — it’s just *continuously bumping* the break pointer, requesting *more, and more* memory from OS. At some point we’ll just run out of memory, and won’t be able to satisfy an allocation request. What we need here is an ability to *reuse* previously freed blocks.

Note: the allocator from above is known as **Sequential allocator** (aka the **“Bump”-allocator**). Yes, it’s that trivial, and just constantly bumping the allocation pointer until it reaches the “end of the heap”, at which point a GC is called, which reclaims the allocation area, relocating the objects around. Below we implement a **Free-list allocator**, which can reuse the blocks right away.

We’ll first look at objects freeing, and then will improve the allocator by adding the reuse algorithm.

Freeing the objects

The free procedure is quite simple and just sets the used flag to false:

```
1/**
2 * Frees a previously allocated block.
3 */
4void free(word_t *data) {
5  auto block = getHeader(data);
6  block->used = false;
7}
```

The free function receives an actual *user pointer* from which it obtains the corresponding Block, and updates the used flag.

The corresponding test:

```
1...
2
```

```
3// ------------------------------------
4// Test case 3: Free the object
5//
6
7free(p2);
8assert(p2b->used == false);
```

Plain simple. OK, now let's finally implement the *reuse* functionality, so we don't exhaust quickly all the available OS memory.

Blocks reuse

Our free function *doesn't* actually return (unmap) the memory back to OS, it just sets the used flag to false. This means we can (read: should!) *reuse* the free blocks in future allocations.

Our alloc function is updated as follows:

```
1 /**
2  * Allocates a block of memory of (at least)
3 `size` bytes.
4  */
5 word_t *alloc(size_t size) {
6   size = align(size);
7
8   // ------------------------------------------
9 -------------
10  // 1. Search for an available free block:
11
12  if (auto block = findBlock(size))
13{                   // (1)
14    return block->data;
15  }
16
17  // ------------------------------------------
18-------------
19  // 2. If block not found in the free list,
20request from OS:
21
22  auto block = requestFromOS(size);
23
```

```
  block->size = size;
  block->used = true;

  // Init heap.
  if (heapStart == nullptr) {
    heapStart = block;
  }

  // Chain the blocks.
  if (top != nullptr) {
    top->next = block;
  }

    top = block;

    // User payload:
    return block->data;
  }
```

The actual reuse functionality is managed in (1) by the findBlock function. Let's take a look at its implementation, starting from the *first-fit* algorithm.

First-fit search

The findBlock procedure should try to find a block of an appropriate size. This can be done in several ways. The first one we consider is the *first-fit* search.

The first-fit algorithm traverses all the blocks starting at the beginning of the heap (the heapStart which we initialized on first allocation). It returns the first found block if it *fits the size*, or the nullptr otherwise.

```
/**
 * First-fit algorithm.
 *
 * Returns the first free block which fits the
size.
 */
```

```
7 Block *firstFit(size_t size) {
8   auto block = heapStart;
9
10  while (block != nullptr) {
11    // O(n) search.
12    if (block->used || block->size < size) {
13      block = block->next;
14      continue;
15    }
16
17    // Found the block:
18    return block;
19  }
20
21  return nullptr;
22}
23
24/**
25 * Tries to find a block of a needed size.
26 */
27Block *findBlock(size_t size) {
28  return firstFit(size);
  }
```

Here is a picture of the first-fit result:

Figure 5. First-fit search

The first found block is returned, even if it's *much larger* in size than requested. We'll fix this below with the next- and best-fit allocations.

The assertion test case:

```
1 // ------------------------------------
2 // Test case 4: The block is reused
3 //
4
```

```
5 // A consequent allocation of the same size
6 reuses
7 // the previously freed block.
8 //
9
10auto p3 = alloc(8);
11auto p3b = getHeader(p3);
  assert(p3b->size == 8);
  assert(p3b == p2b);  // Reused!
```

Make sure all the assertions still pass. Now let's see how we can improve the search of the blocks. We consider now the *next-fit* algorithm, which will be your first assignment.

Assignments

Below are the assignments you'll need to implement in order to complete and improve this allocator.

Next-fit search

The *next-fit* is a variation of the first-fit, however it continues the next search from the *previous* successful position. This allows *skipping* blocks of a *smaller size* at the beginning of the heap, so you don't have to traverse them constantly getting frequent requests for larger blocks. When it reaches the end of the list it starts over from the beginning, and so is sometimes called *circular first-fit allocation*.

Again the difference from the basic first-fit: say you have 100 blocks of size 8 at the beginning of the heap, which are followed by the blocks of size 16. Each alloc(16) request would result in traversing all those 100 blocks at the beginning of the heap in first-fit, while in the next-fit it'll start from the previous successful position of a 16-size block.

Here is a picture example:

Figure 6. Next-fit search

On the subsequent request for a larger block, we find it right away, skipping all the smaller blocks at the beginning.

Let’s define the search mode, and also helper methods to reset the heap:

```
1 /**
2  * Mode for searching a free block.
3  */
4 enum class SearchMode {
5   FirstFit,
6   NextFit,
7 };
8
9 /**
10 * Previously found block. Updated in `nextFit`.
11 */
12static Block *searchStart = heapStart;
13
14/**
15 * Current search mode.
16 */
17static auto searchMode = SearchMode::FirstFit;
18
19/**
20 * Reset the heap to the original position.
21 */
22void resetHeap() {
23  // Already reset.
24  if (heapStart == nullptr) {
25    return;
26  }
27
28  // Roll back to the beginning.
29  brk(heapStart);
```

```
30
31  heapStart = nullptr;
32  top = nullptr;
33  searchStart = nullptr;
34}
35
36/**
37 * Initializes the heap, and the search mode.
38 */
39void init(SearchMode mode) {
40  searchMode = mode;
41  resetHeap();
42}
```

Go ahead, and implement the next-fit algorithm:

```
1 ...
2
3 /**
4  * Next-fit algorithm.
5  *
6  * Returns the next free block which fits the
7 size.
8  * Updates the `searchStart` of success.
9  */
10Block *nextFit(size_t size) {
11  // Implement here...
12}
13
14/**
15 * Tries to find a block that fits.
16 */
17Block *findBlock(size_t size) {
18  switch (searchMode) {
19    case SearchMode::FirstFit:
20      return firstFit(size);
21    case SearchMode::NextFit:
22      return nextFit(size);
23  }
  }
```

And here is a test-case for it:

```
1 ...
2
3 // Init the heap, and the searching algorithm.
4 init(SearchMode::NextFit);
5
6 // -------------------------------------
7 // Test case 5: Next search start position
8 //
9
10// [[8, 1], [8, 1], [8, 1]]
11alloc(8);
12alloc(8);
13alloc(8);
14
15// [[8, 1], [8, 1], [8, 1], [16, 1], [16, 1]]
16auto o1 = alloc(16);
17auto o2 = alloc(16);
18
19// [[8, 1], [8, 1], [8, 1], [16, 0], [16, 0]]
20free(o1);
21free(o2);
22
23// [[8, 1], [8, 1], [8, 1], [16, 1], [16, 0]]
24auto o3 = alloc(16);
25
26// Start position from o3:
27assert(searchStart == getHeader(o3));
28
29// [[8, 1], [8, 1], [8, 1], [16, 1], [16, 1]]
30//                           ^ start here
31alloc(16);
```

OK, the next-fit might be better, however it still would return an *oversized* block, even if there are other blocks which *fit better*. Let's take a look at the *best-fit* algorithm, which solves this issue.

Best-fit search

The main idea of the *best-fit* search, is to try finding a block which fits *the best*.

For example, having blocks of [4, 32, 8] sizes, and the request for alloc(8), the first- and the next-fit searches would return the second block of size 32, which *unnecessary* wastes the space. Obviously, returning the third block of size 8 would fit *the best* here.

Note: below we're going to implement *blocks splitting*, and the block of size 32 from above would result in splitting it in two blocks, taking only 8 requested bytes for the first one.

Here is a picture example:

Figure 7. Best-fit search

The first free block is skipped here, being *too large*, so the search results in the second block.

```
1 /**
2  * Mode for searching a free block.
3  */
4 enum class SearchMode {
5   ...
6   BestFit,
7 };
8
9 ...
10
11/**
12 * Best-fit algorithm.
13 *
14 * Returns a free block which size fits the best.
15 */
16Block *bestFit(size_t size) {
17  // Implement here...
18}
19
```

```
/**
 * Tries to find a block that fits.
 */
Block *findBlock(size_t size) {
  switch (searchMode) {
    ...
    case SearchMode::BestFit:
      return bestFit(size);
  }
}
```

And here is a test case:

```
...

init(SearchMode::BestFit);

// ------------------------------------
// Test case 6: Best-fit search
//

// [[8, 1], [64, 1], [8, 1], [16, 1]]
alloc(8);
auto z1 = alloc(64);
alloc(8);
auto z2 = alloc(16);

// Free the last 16
free(z2);

// Free 64:
free(z1);

// [[8, 1], [64, 0], [8, 1], [16, 0]]

// Reuse the last 16 block:
auto z3 = alloc(16);
assert(getHeader(z3) == getHeader(z2));

// [[8, 1], [64, 0], [8, 1], [16, 1]]

// Reuse 64, splitting it to 16, and 48
```

```
30z3 = alloc(16);
31assert(getHeader(z3) == getHeader(z1));
32
33// [[8, 1], [16, 1], [48, 0], [8, 1], [16, 1]]
```

OK, looks better! However, even with the best-fit we may end up returning a block which is *larger* than the requested. We should now look at another optimization, known as *blocks splitting*.

Blocks splitting

Currently if we found a block of a *suitable size*, we just use it. This might be inefficient in case if a found block is *much larger* than the requested one.

We shall now implement a procedure of *splitting* a larger free block, taking from it only a smaller chunk, which is requested. The other part *stays free*, and can be used in further allocation requests.

Here is an example:

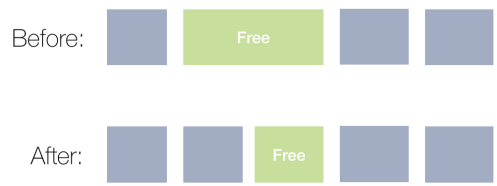

Figure 8. Block split

Our previous first-fit algorithm would just took the whole large block, and we would be out of free blocks in the list. With the blocks splitting though, we still have a free block to reuse.

The listAllocate function is called whenever a block is found in the searching algorithm. It takes care of splitting a larger block on smaller:

```
1 /**
2  * Splits the block on two, returns the pointer
3 to the smaller sub-block.
4  */
```

```
5 Block *split(Block *block, size_t size) {
6   // Implement here...
7 }
8
9 /**
10 * Whether this block can be split.
11 */
12inline bool canSplit(Block *block, size_t size) {
13  // Implement here...
14}
15
16/**
17 * Allocates a block from the list, splitting if
18needed.
19 */
20Block *listAllocate(Block *block, size_t size) {
21  // Split the larger block, reusing the free
22part.
23  if (canSplit(block, size)) {
24    block = split(block, size);
25  }
26
27  block->used = true;
28  block->size = size;

    return block;
  }
```

With splitting we have now good reuse of the free space. Yet however, if we have *two fee adjacent blocks* on the heap of size 8, and a user requests a block of size 16, we *can't* satisfy an allocation request. Let's take a look at another optimization for this – the *blocks coalescing*.

Blocks coalescing

On *freeing* the objects, we can do the *opposite* to splitting operation, and *coalesce* two (or more) *adjacent blocks* to a larger one.

Here is an example:

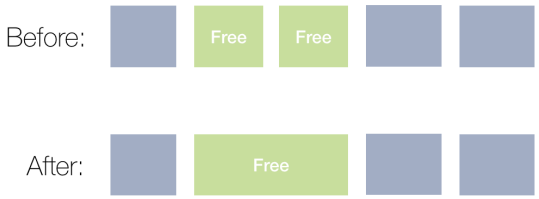

Figure 9. Blocks coalescing

Go ahead and implement the merging procedure:

```
1 ...
2
3 /**
4  * Whether we should merge this block.
5  */
6 bool canCoalesce(Block *block) {
7   return block->next && !block->next->used;
8 }
9
10/**
11 * Coalesces two adjacent blocks.
12 */
13Block *coalesce(Block *block) {
14  // Implement here...
15}
16
17/**
18 * Frees the previously allocated block.
19 */
20void free(word_t *data) {
21  auto block = getHeader(data);
22  if (canCoalesce(block)) {
23    block = coalesce(block);
24  }
25  block->used = false;
26}
```

Notice that we coalesce only with the *next* block, since we only have access to the next pointer. Experiment with adding prev pointer to the object header, and consider coalescing with the previous block as well.

Blocks coalescing allows us satisfying allocation requires for larger blocks, however, we still have to traverse the list with O(n) approach in order to find a free block. Let's see how we can fix this with the *explicit free-list* algorithm.

Explicit Free-list

Currently we do a linear search analyzing each block, *one by one*. A more efficient algorithm would be using an *explicit free-list*, which links *only* free blocks.

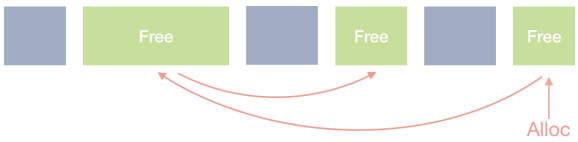

Figure 10. Explicit Free-list

This might be a *significant* performance improvement, when the heap is getting larger, and one needs to traverse a lot of objects in the basic algorithms.

An explicit free-list can be implemented directly in the object header. For this the next, and prev pointers would point to the *free blocks*. Procedures of splitting and coalescing should be updated accordingly, since next, and prev won't be pointing to adjacent blocks anymore.

Alternatively, one could just have a separate additional free_list data structure (std::list would work):

```
1#include <list>
2
3/**
4 * Free list structure. Blocks are added to the
5free list
6 * on the `free` operation. Consequent allocations
7of the
8 * appropriate size reuse the free blocks.
  */
```

```
 static std::list<Block *> free_list;
```

The alloc procedure would just traverse the free_list: if nothing is found, request from OS:

```
/**
 * Mode for searching a free block.
 */
enum class SearchMode {
  ...
  FreeList,
};

/**
* Explicit free-list algorithm.
*/
Block *freeList(size_t size) {
  for (const auto &block : free_list) {
    if (block->size < size) {
      continue;
    }
    free_list.remove(block);
    return listAllocate(block, size);
  }
  return nullptr;
}

/**
* Tries to find a block that fits.
*/
Block *findBlock(size_t size) {
  switch (searchMode) {
    ...
    case SearchMode::FreeList:
      return freeList(size);
  }
}
```

The free operation would just put the block into the free_list:

```
/**
* Frees the previously allocated block.
```

```
3 */
4void free(word_t *data) {
5  ...
6  if (searchMode == SearchMode::FreeList) {
7    free_list.push_back(block);
8  }
9}
```

Experiment and implement a doubly-linked list approach, using next, and prev in the header for the free-list purposes.

Finally let's consider one more speed optimization, which brings much faster search for blocks of *predefined sizes*. The approach is knows as *segregated list*.

Segregated-list search

Considered above searching algorithms (first-fit, next-fit, and the best-fit) do a linear search through the blocks of *different sizes*. This slows down the search of a block of an appropriate size. The idea of the *segregated-fit* is to *group* the blocks *by size*.

That is, instead of *one* list of blocks, we have *many* lists of blocks, but each list contains *only* blocks of a certain size.

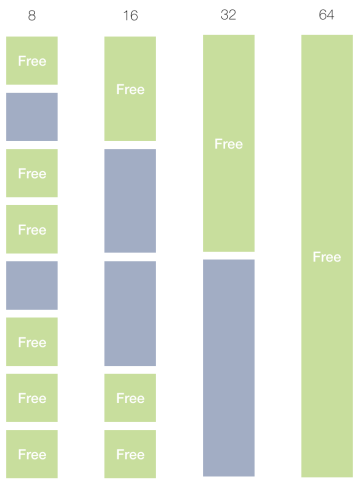

Figure 11. Segregated list

In this example the first group contains blocks *only* of size 8, the second group – of size 16, third – 32, etc. Then, if we get a request to allocate 16 bytes, we directly *jump* to the needed bucket (group), and allocate from there.

There are several implementations of the heap segregation. One can pre-allocate a large arena, and split it on buckets, having a *contiguous array of buckets*, with fast jumps to needed group and a block within a group.

Another approach is to have just an *array of lists*. Example below initializes these segregated lists. When a first block of the appropriate size is allocated, it replaces the nullptr value in the bucket. Next allocation of the same size chains it further.

That is, instead of one heapStart, and top, we have many “heapStarts”, and “tops”

```
1 /**
2  * Segregated lists. Reserve the space
3  * with nullptr. Each list is initialized
4  * on first allocation.
5  */
6 Block *segregatedLists[] = {
7   nullptr,   //   8
8   nullptr,   //  16
9   nullptr,   //  32
10  nullptr,   //  64
11  nullptr,   // 128
12  nullptr,   // any > 128
13};
```

And the actual search within a bucket can reuse any of the considered above algorithms. In the example below we reuse firstFit search.

```
1 enum class SearchMode {
2   ...
3   SegregatedList,
4 };
```

```
5
6 ...
7
8 /**
9  * Gets the bucket number from segregatedLists
10 * based on the size.
11 */
12inline int getBucket(size_t size) {
13  return size / sizeof(word_t) - 1;
14}
15
16/**
17 * Segregated fit algorithm.
18 */
19Block *segregatedFit(size_t size) {
20  // Bucket number based on size.
21  auto bucket = getBucket(size);
22  auto originalHeapStart = heapStart;
23
24  // Init the search.
25  heapStart = segregatedLists[bucket];
26
27  // Use first-fit here, but can be any:
28  auto block = firstFit(size);
29
30  heapStart = originalHeapStart;
31  return block;
32}
33
34/**
35 * Tries to find a block that fits.
36 */
37Block *findBlock(size_t size) {
38  switch (searchMode) {
39    ...
40    case SearchMode::SegregatedList:
41      return segregatedFit(size);
42  }
43}
```

OK, we have optimized the search speed. Now let's see how we can optimize the storage size.

Optimizing the storage

Currently object header is pretty heavy: it contains size, used, and next fields. One can encode some of the information more efficiently.

For example, instead of explicit next pointer, we can get access to the next block by simple *size* calculation of the block.

In addition, since our blocks are aligned by the word size, we can reuse LSB (least-significant bits) of the size field for our purposes. For example: 1000 – size 8, and is *not* used. And the 1001 – size 8, and *is* used.

The header structure would look as follows:

```
1 struct Block {
2   // -----------------------------------
3   // 1. Object header
4
5   /**
6    * Object header. Encodes size and used flag.
7    */
8   size_t header;
9
10  // -----------------------------------
11  // 2. User data
12
13  /**
14   * Payload pointer.
15   */
16  word_t data[1];
17};
```

And couple of encoding/decoding functions:

```
1 /**
2  * Returns actual size.
3  */
4 inline size_t getSize(Block *block) {
```

```
5   return block->header & ~1L;
6 }
7
8 /**
9  * Whether the block is used.
10 */
11inline bool isUsed(Block *block) {
12  return block->header & 1;
13}
14
15/**
16 * Sets the used flag.
17 */
18inline void setUsed(Block *block, bool used) {
19  if (used) {
20    block->header |= 1;
21  } else {
22    block->header &= ~1;
23  }
24}
```

This is basically trading storage for speed. In case when you have a smaller storage, you may consider such encoding/decoding. And vice-versa, when you have a lot of storage, and the speed matters, using explicit fields in structure might be faster, than constant encoding and decoding of the object header data.

Custom sbrk

As was mentioned above, on Mac OS sbrk is currently deprecated and is emulated via mmap.

(Wait, so if sbrk is deprecated, why did we use it?)

What is deprecated is the specific function, and on specific OS. However by itself, the sbrk is not just a “function”, it’s the *mechanism*, an *abstraction*. As mentioned, it also directly corresponds to the *bump-allocator*.

Production malloc implementations also use mmap instead of brk, when a larger chunk of memory is

requested. In general mmap can be used for both, mapping of *external files* to memory, and also for creating *anonymous mappings* (not backed by any file). The later is exactly used for memory allocation.

Example:

```
#include <sys/mman.h>
...

/**
 * Allocation arena for custom sbrk example.
 */
static void *arena = nullptr;

/**
* Program break;
*/
static char *_brk = nullptr;

/**
* Arena size.
*/
static size_t arenaSize = 4194304;  // 4 MB

...

// Map a large chunk of anonymous memory, which is a
// virtual heap for custom sbrk manipulation.
arena = mmap(0, arenaSize, PROT_READ | PROT_WRITE,
             MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
```

Instead of using the standard sbrk, implement own one:

```
void *_sbrk(intptr_t increment) {
  // Implement here...

  // 0. Pre-allocate a large arena using `mmap`.
Init program break
```

```
6   //    to the beginning of this arena.
7
8   // 1. If `increment` is 0, return current break
9 position.
10
11  // 2. If `current + increment` exceeds the top
12of
13  //    the arena, return -1.
14
    // 3. Otherwise, increase the program break on
    //    `increment` bytes.
  }
```

Conclusion

At this point we're finishing our discussion about memory allocators, and in the next lectures will be working already with *garbage collection* algorithms. As we will see, different collectors work with different allocators, and designing a memory manager requires choosing a correct correspondence between the collector, and allocator.

Again, you can enroll to full course here:

Try solving all the assignments yourself first. If you're stuck, you can address the full source code with the solutions here.

To summarize our lecture:

- **Sequential (Bump) allocator** is *the fastest*, and is similar to the *stack* allocation: just continuously bump the allocation pointer. Unfortunately, not every programming language can allow usage of the Bump-allocator. In particular, C/C++, which expose the pointers semantics, cannot *relocate* objects on GC cycle, and therefore has to use slower Free-list allocator
- Bump-allocator is used in systems with *Mark-Compact*, *Copying*, and *Generational collectors*

- **Free-list allocator** is slower, and traverses the *linked list* of blocks, until it finds a block of a needed size. This traversal is the main reason why the heap allocation is in general considered slower than the stack allocation (which is not true in case of the Bump-allocator)
- Free-list allocator (such as malloc) is used in systems with *Mark-Sweep*, *Reference counting* collectors, and also in systems with *manual* memory management
- List search can be implemented as **first-fit**, **next-fit**, and **best-fit** algorithms
- Even faster list search can be achieved by using an **explicit free-list**
- And even more faster search can be achieved in the **segregated free-lists**
- Encoding object header, consider trading speed for storage, and vice-versa

If you have any questions, or suggestions, as always I'll be glad to discuss them in comments. And see you in the class!

Written by: Dmitry Soshnikov
Published on: Feb 6th, 2019