

MMU initialisation

First off, we start by loading the needed modules - `mmu_mod`, `frames_mod` and `heap_mod`.

`mmu_mod` will allocate and map the first chunk of memory to use. It does so without knowing much about memory allocation - just takes a first fit chunk from the physical memory map, passed to us by the bootloader. Parts of this memory will be used by `frames_mod` and `heap_mod` as well, so we need a way to know how much memory they would need.

We need `frames_mod` to tell us, how much it needs to manage the physical memory, for the heap we just assume some size that we would need dynamically during rest of startup.

```
int required = frames_mod->required_size();
int initial_heap_size = 128*KiB;
```

How `frames` module figures out its required space we will understand in a moment, when we see the structures it uses for bookkeeping.

With space requirements in our hands we can start initialising MMU structures.

MMU state is pretty large, it consists of page directory and page tables used for initial mappings, RAM table and some important closures - system frame allocator, heap and stretch allocator, as well as `ramtab` closure.

MMU create method will initialise all these structures in one big chunk of memory allocated from physical memory map. It also means this data may overwrite some of the information loaded by grub (namely the bootimage), but we've already copied everything we need higher in memory while loading the modules, so this is fine.

After allocating all page structures, we will fill them up by means of `enter_mappings()` function. This function will cover allocated memory and also allocate extra L2 page tables. Finally, all important pointers are set up and the CPU is switched to the newly allocated page directory.

Next step is to create the physical frame allocator.

Frame allocator initialisation

Frame allocator needs to track all frames of physical memory, their availability and ownership.

It does so by keeping a linked list of allocation regions (which roughly correspond to areas in the BIOS memory map passed to us). Each region entry has a list of frames in this region and their availability.

```
struct frames_module_v1_state
{
    address_t start;
    size_t n_logical_frames;
```

```

uint32_t frame_width;
memory_v1_attrs attrs;
ramtab_v1_closure* ramtab;
frames_module_v1_state* next;
frame_st* frames;
};

```

Each region starts at address “start” and lasts for “n_logical_frames”, each of $2^{\text{frame_width}}$ bytes in size. Memory in this region has attributes “attrs” and if this is RAM, it has an associated “ramtab”.

“Next” points to the next region in the list and “frames” points to an array of n_logical_frames entries describing which frames are free. This array consists of very simple entries:

```

struct frame_st
{
    uint32_t free;
};

```

The “free” member indicates how many frames starting from a given frame index are free. Zero means this frame is occupied, any N above zero means that this frame, and N-1 frames after it are available.

It makes it fairly easy to look for the first-fit or the best-fit frame stretches. On the downside you need to update multiple frames at once if you allocate some frames at the end, so the best strategy for physical memory allocation is to allocate frames at the start (just set the value of “free” to 0) and to release them from end to start (so the last frame of the freed region takes value of “free” from the next frame plus one, and so on).

Frame allocator also keeps track of domains that request frame allocation, or clients.

For each new client, there’s a structure describing it and the contract obligations that frame allocator has for this client.

```

struct frame_allocator_v1_state
{
    frame_allocator_v1_closure closure;

    uint32_t n_allocated_phys_frames;

    uint32_t owner;
    size_t guaranteed_frames;
    size_t extra_frames;

    heap_v1_closure* heap;
    frames_module_v1_state* module_state;
};

```

This structure describes the owner (by storing its domain record’s physical address in “owner”), amount of already allocated frames, guaranteed total amount of frames that this client could allocate and amount of extra frames that client might receive unless there’s memory pressure. It also contains a pointer to the region list used to allocate frames in “module_state”. Multiple clients may point to the same list for memory allocation.

Whenever a frame is occupied and taken from that list, owner information is recorded inside the ramtab.

The ramtab

Ramtab registers ownership information for allocated frames, it also records if these frames are “nailed” - that is, cannot be unmapped or released.

```
struct ramtab_entry_t
{
    address_t owner;
    uint16_t frame_width;
    uint16_t state;
} PACKED;
```

That’s about all it does at the moment, so I won’t focus on it.

The heap

After all the above the heap looks fairly simplistic. There’s a header record with pointers to about 40 free lists - for blocks of different sizes. These free lists provide means to quickly allocate blocks of approximately requested size (or “best fit”).

```
static const int SMALL_BLOCKS = 16;
static const int LARGE_BLOCKS = 24;
static const int COUNT = (SMALL_BLOCKS + LARGE_BLOCKS + 1);
static const memory_v1_size all_sizes[COUNT];

heap_rec_t* blocks[COUNT];
```

The structure is two-dimensional: free lists connect memory blocks through “next” pointers, while allocated blocks are connected by their physical adjacency.

Each block has a header:

```
struct heap_rec_t
{
    memory_v1_size prev; // either a magic or size of previous block (backlink).
    memory_v1_size size; // size of allocated block, including the end footer.
    uint32_t index; // allocation table index.
    union {
        heap_t* heap; // when busy
        heap_rec_t* next; // when free
    };
};
```

The end footer of the block is actually the first field of the next block, which contains a previous block link in case the block is free (this makes free lists double-linked lists actually) or HEAP_MAGIC in case the block is occupied. Allocation table index allows for quick return of freed blocks into their corresponding free lists.

To allocate memory we round up requested block size to a minimum granularity we support (8 bytes in current implementation), then find index of a corresponding free list using find_index() call and look if there’s a free block available.

If it is - we set up some meta information (like the owning heap and the fact that the block is now used) and return.

If there isn't free block available - we take some space from the "all sizes" block, which is usually just huge chunk of still available memory.

This memory can get fragmented as well, and when there's no more memory available heap must grow. The current "raw" heap does not support this, and it will be implemented for stretch-backed heaps later, when stretch allocator works - which is the topic for the next post. Stay tuned!