Chapter 10

Page Frame Allocation

When using virtual memory, how does the OS know which parts of memory are free to use? That is the role of the page frame allocator.

Managing Available Memory

How Much Memory is There?

First we need to know how much memory is available on the computer the OS is running on. The easiest way to do this is to read it from the multiboot structure [19] passed to us by GRUB. GRUB collects the information we need about the memory - what is reserved, I/O mapped, read-only etc. We must also make sure that we don't mark the part of memory used by the kernel as free (since GRUB doesn't mark this memory as reserved). One way to know how much memory the kernel uses is to export labels at the beginning and the end of the kernel binary from the linker script:

```
ENTRY(loader)           /* the name of the entry symbol */

. = 0xC0100000          /* the code should be relocated to 3 GB + 1 MB */

/* these labels get exported to the code files */
kernel_virtual_start = .;
kernel_physical_start = . - 0xC0000000;

/* align at 4 KB and load at 1 MB */
.text ALIGN (0x1000) : AT(ADDR(.text)-0xC0000000)
{
    *(.text)            /* all text sections from all files */
}

/* align at 4 KB and load at 1 MB + . */
.rodata ALIGN (0x1000) : AT(ADDR(.rodata)-0xC0000000)
{
    *(.rodata*)         /* all read-only data sections from all files */
}
```

```
/* align at 4 KB and load at 1 MB + . */
.data ALIGN (0x1000) : AT(ADDR(.data)-0xC0000000)
{
    *(.data)            /* all data sections from all files */
}

/* align at 4 KB and load at 1 MB + . */
.bss ALIGN (0x1000) : AT(ADDR(.bss)-0xC0000000)
{
    *(COMMON)           /* all COMMON sections from all files */
    *(.bss)             /* all bss sections from all files */
}

kernel_virtual_end = .;
kernel_physical_end = . - 0xC0000000;
```

These labels can directly be read from assembly code and pushed on the stack to make them available to C code:

```
extern kernel_virtual_start
extern kernel_virtual_end
extern kernel_physical_start
extern kernel_physical_end

; ...

push kernel_physical_end
push kernel_physical_start
push kernel_virtual_end
push kernel_virtual_start

call kmain
```

This way we get the labels as arguments to `kmain`. If you want to use C instead of assembly code, one way to do it is to declare the labels as functions and take the addresses of these functions:

```
void kernel_virtual_start(void);

/* ... */

unsigned int vaddr = (unsigned int) &kernel_virtual_start;
```

If you use GRUB modules you need to make sure the memory they use is marked as reserved as well.

Note that the available memory does not need to be contiguous. In the first 1 MB there are several I/O-mapped memory sections, as well as memory used by GRUB and the BIOS. Other parts of the memory might be similarly unavailable.

It's convenient to divide the memory sections into complete page frames, as we can't map part of pages into memory.

Managing Available Memory

How do we know which page frames are in use? The page frame allocator needs to keep track of which are free and which aren't. There are several ways to do this: bitmaps, linked lists, trees, the Buddy System (used by Linux) etc. For more information about the different algorithms see the article on OSDev [38].

Bitmaps are quite easy to implement. One bit is used for each page frame and one (or more) page frames are dedicated to store the bitmap. (Note that this is just one way to do it, other designs might be better and/or more fun to implement.)

How Can We Access a Page Frame?

The page frame allocator returns the physical start address of the page frame. This page frame is not mapped in - no page table points to this page frame. How can we read and write data to the frame?

We need to map the page frame into virtual memory, by updating the PDT and/or PT used by the kernel. What if all available page tables are full? Then we can't map the page frame into memory, because we'd need a new page table - which takes up an entire page frame - and to write to this page frame we'd need to map its page frame... Somehow this circular dependency must be broken.

One solution is to reserve a part of the first page table used by the kernel (or some other higher-half page table) for temporarily mapping page frames to make them accessible. If the kernel is mapped at `0xC0000000` (page directory entry with index 768), and 4 KB page frames are used, then the kernel has at least one page table. If we assume - or limit us to - a kernel of size at most 4 MB minus 4 KB we can dedicate the last entry (entry 1023) of this page table for temporary mappings. The virtual address of pages mapped in using the last entry of the kernel's PT will be:

```
(768 << 22) | (1023 << 12) | 0x000 = 0xC03FF000
```

After we've temporarily mapped the page frame we want to use as a page table, and set it up to map in our first page frame, we can add it to the paging directory, and remove the temporary mapping.

A Kernel Heap

So far we've only been able to work with fixed-size data, or directly with raw memory. Now that we have a page frame allocator we can implement `malloc` and `free` to use in the kernel.

Kernighan and Ritchie [8] have an example implementation in their book [8] that we can draw inspiration from. The only modification we need to do is to replace calls to `sbrk/brk` with calls to the page frame allocator when more memory is needed. We must also make sure to map the page frames returned by the page frame allocator to virtual addresses. A correct implementation should also return page frames to the page frame allocator on call to `free`, whenever sufficiently large blocks of memory are freed.

Further reading

- The OSDev wiki page on page frame allocation: http://wiki.osdev.org/Page_Frame_Allocation