Chapter 9

Paging

Segmentation translates a logical address into a linear address. *Paging* translates these linear addresses onto the physical address space, and determines access rights and how the memory should be cached.

Why Paging?

Paging is the most common technique used in x86 to enable virtual memory. Virtual memory through paging means that each process will get the impression that the available memory range is `0x00000000 - 0xFFFFFFFF` even though the actual size of the memory might be much less. It also means that when a process addresses a byte of memory it will use a virtual (linear) address instead of physical one. The code in the user process won't notice any difference (except for execution delays). The linear address gets translated to a physical address by the MMU and the page table. If the virtual address isn't mapped to a physical address, the CPU will raise a page fault interrupt.

Paging is optional, and some operating systems do not make use of it. But if we want to mark certain areas of memory accessible only to code running at a certain privilege level (to be able to have processes running at different privilege levels), paging is the neatest way to do it.

Paging in x86

Paging in x86 (chapter 4 in the Intel manual [33]) consists of a *page directory* (PDT) that can contain references to 1024 *page tables* (PT), each of which can point to 1024 sections of physical memory called *page frames* (PF). Each page frame is 4096 byte large. In a virtual (linear) address, the highest 10 bits specifies the offset of a page directory entry (PDE) in the current PDT, the next 10 bits the offset of a page table entry (PTE) within the page table pointed to by that PDE. The lowest 12 bits in the address is the offset within the page frame to be addressed.

All page directories, page tables and page frames need to be aligned on 4096 byte addresses. This makes it possible to address a PDT, PT or PF with just the highest 20 bits of a 32 bit address, since the lowest 12 need to be zero.

The PDE and PTE structure is very similar to each other: 32 bits (4 bytes), where the highest 20 bits points to a PTE or PF, and the lowest 12 bits control access rights and other configurations. 4 bytes times 1024 equals 4096 bytes, so a page directory and page table both fit in a page frame themselves.

The translation of linear addresses to physical addresses is described in the figure below.

While pages are normally 4096 bytes, it is also possible to use 4 MB pages. A PDE then points directly to a 4 MB page frame, which needs to be aligned on a 4 MB address boundary. The address translation is almost the same as in the figure, with just the page table step removed. It is possible to mix 4 MB and 4 KB pages.

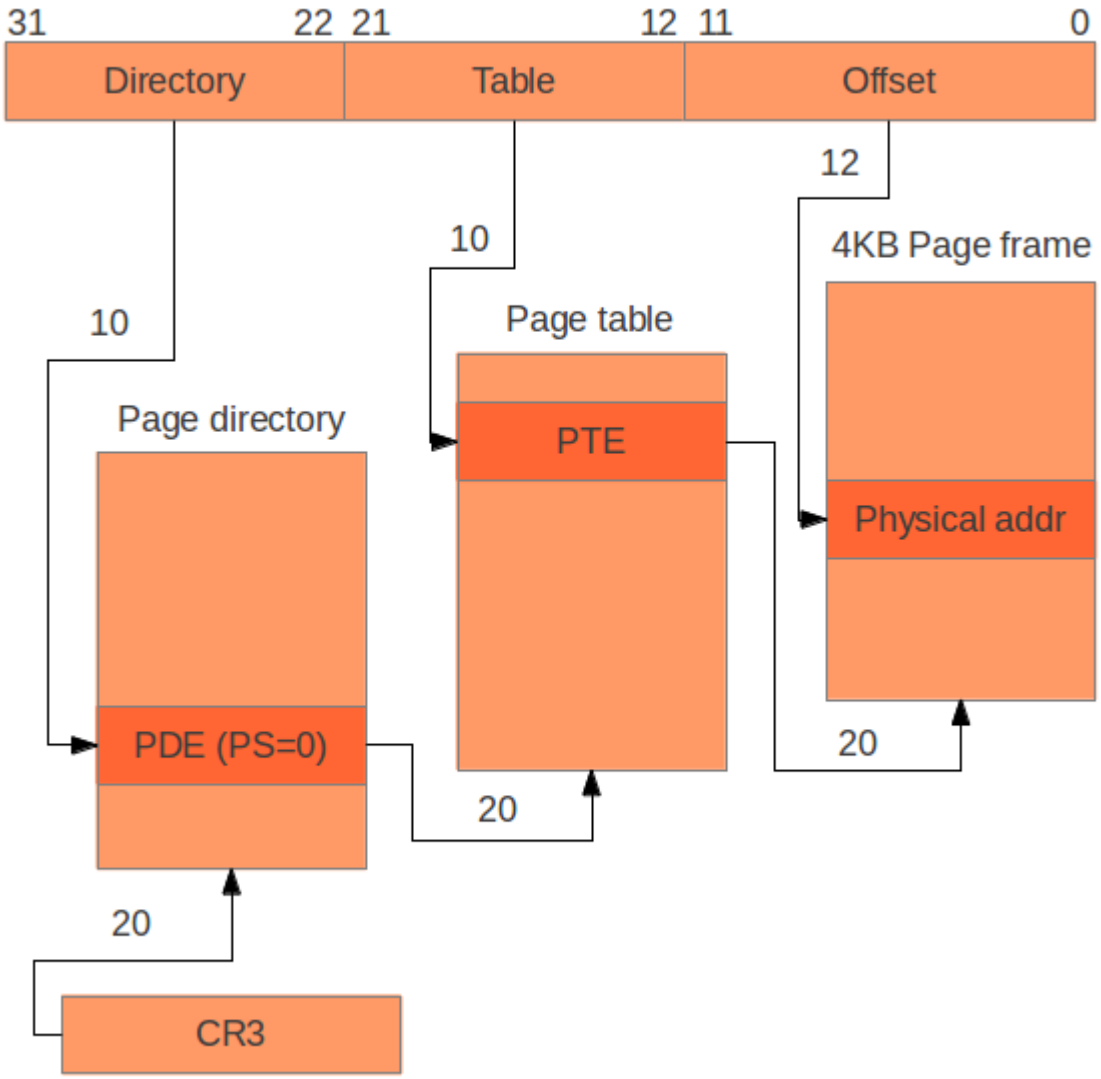

Figure 9.1: Translating virtual addresses (linear addresses) to physical addresses.

The 20 bits pointing to the current PDT is stored in the register `cr3`. The lower 12 bits of `cr3` are used for configuration.

For more details on the paging structures, see chapter 4 in the Intel manual [33]. The most interesting bits are *U/S*, which determine what privilege levels can access this page (PL0 or PL3), and *R/W*, which makes the memory in the page read-write or read-only.

Identity Paging

The simplest kind of paging is when we map each virtual address onto the same physical address, called *identity paging*. This can be done at compile time by creating a page directory where each entry points to its corresponding 4 MB frame. In NASM this can be done with macros and commands (`%rep`, `times` and `dd`). It can of course also be done at run-time by using ordinary assembly code instructions.

Enabling Paging

Paging is enabled by first writing the address of a page directory to `cr3` and then setting bit 31 (the PG "paging-enable" bit) of `cr0` to `1`. To use 4 MB pages, set the PSE bit (Page Size Extensions, bit 4) of `cr4`. The following assembly code shows an example:

```
; eax has the address of the page directory
mov cr3, eax

mov ebx, cr4        ; read current cr4
or  ebx, 0x00000010 ; set PSE
mov cr4, ebx        ; update cr4

mov ebx, cr0        ; read current cr0
or  ebx, 0x80000000 ; set PG
mov cr0, ebx        ; update cr0

; now paging is enabled
```

A Few Details

It is important to note that all addresses within the page directory, page tables and in `cr3` need to be physical addresses to the structures, never virtual. This will be more relevant in later sections where we dynamically update the paging structures (see the chapter "User Mode").

An instruction that is useful when an updating a PDT or PT is `invlpg`. It invalidates the *Translation Lookaside Buffer* (TLB) entry for a virtual address. The TLB is a cache for translated addresses, mapping physical addresses corresponding to virtual addresses. This is only required when changing a PDE or PTE that was previously mapped to something else. If the PDE or PTE had previously been marked as not present (bit 0 was set to 0), executing `invlpg` is unnecessary. Changing the value of `cr3` will cause all entries in the TLB to be invalidated.

An example of invalidating a TLB entry is shown below:

```
; invalidate any TLB references to virtual address 0
invlpg [0]
```

Paging and the Kernel

This section will describe how paging affects the OS kernel. We encourage you to run your OS using identity paging before trying to implement a more advanced paging setup, since it can be hard to debug a malfunctioning page table that is set up via assembly code.

Reasons to Not Identity Map the Kernel

If the kernel is placed at the beginning of the virtual address space - that is, the virtual address space (`0x00000000`, `"size of kernel"`) maps to the location of the kernel in memory - there will be issues when linking the user mode process code. Normally, during linking, the linker assumes that the code will be loaded into the memory position `0x00000000`. Therefore, when resolving absolute references, `0x00000000` will be the base address for calculating the exact position. But if the kernel is mapped onto the virtual address space (`0x00000000`, `"size of kernel"`), the user mode process cannot be loaded at virtual address `0x00000000` - it must be placed somewhere else. Therefore, the assumption from the linker that the user mode process is loaded into memory at position `0x00000000` is wrong. This can be corrected by using a linker script which tells the linker to assume a different starting address, but that is a very cumbersome solution for the users of the operating system.

This also assumes that we want the kernel to be part of the user mode process' address space. As we will see later, this is a nice feature, since during system calls we don't have to change any paging structures to get access to the kernel's code and data. The kernel pages will of course require privilege level 0 for access, to prevent a user process from reading or writing kernel memory.

The Virtual Address for the Kernel

Preferably, the kernel should be placed at a very high virtual memory address, for example `0xC0000000` (3 GB). The user mode process is not likely to be 3 GB large, which is now the only way that it can conflict with the kernel. When the kernel uses virtual addresses at 3 GB and above it is called a *higher-half kernel*. `0xC0000000` is just an example, the kernel can be placed at any address higher than 0 to get the same benefits. Choosing the correct address depends on how much virtual memory should be available for the kernel (it is easiest if all memory above the kernel virtual address should belong to the kernel) and how much virtual memory should be available for the process.

If the user mode process is larger than 3 GB, some pages will need to be swapped out by the kernel. Swapping pages is not part of this book.

Placing the Kernel at `0xC0000000`

To start with, it is better to place the kernel at `0xC0100000` than `0xC0000000`, since this makes it possible to map (`0x00000000`, `0x00100000`) to (`0xC0000000`, `0xC0100000`). This way, the entire range (`0x00000000`, `"size of kernel"`) of memory is mapped to the range (`0xC0000000`, `0xC0000000 + "size of kernel"`).

Placing the kernel at `0xC0100000` isn't hard, but it does require some thought. This is once again a linking problem. When the linker resolves all absolute references in the kernel, it will assume that our kernel is loaded at physical memory location `0x00100000`, not `0x00000000`, since relocation is used in the linker script (see the section "Linking the kernel"). However, we want the jumps to be resolved using `0xC0100000` as base address, since otherwise a kernel jump will jump straight into the user mode process code (remember that the user mode process is loaded at virtual memory `0x00000000`).

However, we can't simply tell the linker to assume that the kernel starts (is loaded) at `0xC01000000`, since we want it to be loaded at the physical address `0x00100000`. The reason for having the kernel loaded at 1 MB is because it can't be loaded at `0x00000000`, since there is BIOS and GRUB code loaded below 1 MB. Furthermore, we cannot assume that we can load the kernel at `0xC0100000`, since the machine might not have 3 GB of physical memory.

This can be solved by using both relocation (`.=0xC0100000`) and the `AT` instruction in the linker script. Relocation specifies that non-relative memory-references should should use the relocation address as base in address calculations. `AT` specifies where the kernel should be loaded into memory. Relocation is done at link

time by GNU ld [37], the load address specified by `AT` is handled by GRUB when loading the kernel, and is part of the ELF format [18].

Higher-half Linker Script

We can modify the first linker script to implement this:

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

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

/* 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(.text)-0xC0000000)
{
    *(.rodata*)         /* all read-only data sections from all files */
}

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

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

Entering the Higher Half

When GRUB jumps to the kernel code, there is no paging table. Therefore, all references to `0xC0100000 + X` won't be mapped to the correct physical address, and will therefore cause a general protection exception (GPE) at the very best, otherwise (if the computer has more than 3 GB of memory) the computer will just crash.

Therefore, assembly code that doesn't use relative jumps or relative memory addressing must be used to do the following:

- Set up a page table.
- Add identity mapping for the first 4 MB of the virtual address space.
- Add an entry for `0xC0100000` that maps to `0x0010000`

If we skip the identity mapping for the first 4 MB, the CPU would generate a page fault immediately after paging was enabled when trying to fetch the next instruction from memory. After the table has been created, an jump can be done to a label to make `eip` point to a virtual address in the higher half:

```
; assembly code executing at around 0x00100000
; enable paging for both actual location of kernel
; and its higher-half virtual location

lea ebx, [higher_half] ; load the address of the label in ebx
jmp ebx                ; jump to the label

higher_half:
    ; code here executes in the higher half kernel
    ; eip is larger than 0xC0000000
    ; can continue kernel initialisation, calling C code, etc.
```

The register `eip` will now point to a memory location somewhere right after `0xC0100000` - all the code can now execute as if it were located at `0xC0100000`, the higher-half. The entry mapping of the first 4 MB of virtual memory to the first 4 MB of physical memory can now be removed from the page table and its corresponding entry in the TLB invalidated with `invlpg [0]`.

Running in the Higher Half

There are a few more details we must deal with when using a higher-half kernel. We must be careful when using memory-mapped I/O that uses specific memory locations. For example, the frame buffer is located at `0x000B8000`, but since there is no entry in the page table for the address `0x000B8000` any longer, the address `0xC00B8000` must be used, since the virtual address `0xC0000000` maps to the physical address `0x00000000`.

Any explicit references to addresses within the multiboot structure needs to be changed to reflect the new virtual addresses as well.

Mapping 4 MB pages for the kernel is simple, but wastes memory (unless you have a really big kernel). Creating a higher-half kernel mapped in as 4 KB pages saves memory but is harder to set up. Memory for the page directory and one page table can be reserved in the `.data` section, but one needs to configure the mappings from virtual to physical addresses at run-time. The size of the kernel can be determined by exporting labels from the linker script [37], which we'll need to do later anyway when writing the page frame allocator (see the chapter "Page Frame Allocation).

Virtual Memory Through Paging

Paging enables two things that are good for virtual memory. First, it allows for fine-grained access control to memory. You can mark pages as read-only, read-write, only for PL0 etc. Second, it creates the illusion of contiguous memory. User mode processes, and the kernel, can access memory as if it were contiguous, and the contiguous memory can be extended without the need to move data around in memory. We can also allow the user mode programs access to all memory below 3 GB, but unless they actually use it, we don't have to assign page frames to the pages. This allows processes to have code located near `0x00000000` and the stack at just below `0xC0000000`, and still not require more than two actual pages.

Further Reading

- Chapter 4 (and to some extent chapter 3) of the Intel manual [33] are your definitive sources for the details about paging.
- Wikipedia has an article on paging: http://en.wikipedia.org/wiki/Paging
- The OSDev wiki has a page on paging: http://wiki.osdev.org/Paging and a tutorial for making a higher-half kernel: http://wiki.osdev.org/Higher_Half_bare_bones
- Gustavo Duarte's article on how a kernel manages memory is well worth a read: http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory
- Details on the linker command language can be found at Steve Chamberlain's website [37].
- More details on the ELF format can be found in this presentation: http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf