Report on exercise #3

Matteo Corain S256654

Laboratory #4 – System and device programming – A.Y. 2018-19

The paging mechanism implemented in the given minimal kernel reflects the basic, 32-bits paging mechanism of the Intel x86 architecture. This is schematically described in the picture below, taken directly from the *Intel Software Developer Manual*:

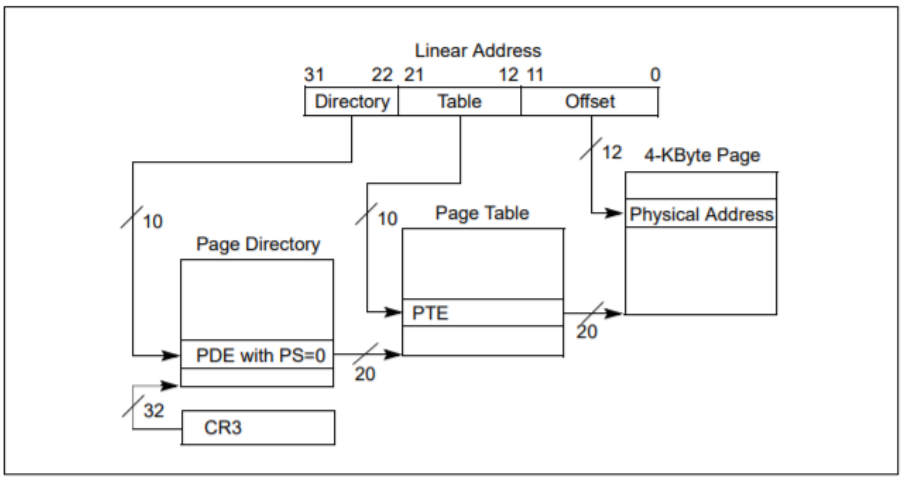

Figure 4-2. Linear-Address Translation to a 4-KByte Page using 32-Bit Paging

In particular, when using this paging mechanism, the memory is split in pages of a fixed dimension, either 4 KB (most common case) or 4 MB. For each page, an entry is stored in a page table, containing the physical address of the frame to which the page is mapped and a bunch of additional information; each page table is capable of storing 1024 entries, each of which is 32 bits long. Page tables are, in turn, grouped in page directories; those are capable as well of storing 1024 entries, each of which is 32 bits long and contains the base address of a page table.

The starting address of the page directory currently in use is stored in the CR3 register; when the access to a certain linear address is requested by a program, the following steps are taken:

- The CPU accesses the current page directory using, as index, the upper part of the linear address of the memory location to be accessed (the first 10 bits, to select among 1024 possible PDEs);
- After having read the base address of the page table to use from the PDE, the CPU accesses the page table using, as index, the following 10 bits of the linear address to retrieve the PTE to use;
- Finally, the CPU sums to the base address of the page read from the PTE the offset value (last 12 bits of the linear address) and accesses the memory at the physical address of the requested location.

In particular, the format of each page table entry is described in the following table:

Bits	Name	Description
0	*P (Present flag)*	It is set if the page is present in the physical memory.
1	*R/W (Read/Write flag)*	It is set if the page is writable, unset if the page is read-only.
2	*U/S (User/Supervisor flag)*	It is set if the page is a user-mode page, unset if it is a supervisor-mode page.
3	*PWT (Page-level Write Through flag)*	It is set if write-through caching should be used for the page, unset if write-back caching is used.

4	*PCD (Page-level Cache Disable)*	It is set if caching should be disabled for the page, unset if caching is enabled.
5	*A (Accessed flag)*	It is set if the page has been accessed since the last refresh.
6	*D (Dirty flag)*	It is set if the page has been modified since the last refresh.
7	*PAT (PAT flag)*	If PAT (Page Attribute Table) is enabled, it determines the memory type used to access the page.
8	*G (Global flag)*	It is set when the translation should be considered global (global translations must be enabled also in the CR4 register).
11:9	*Reserved*	
31:12	*Frame address*	It contains the upper part (first 20 bits, shifted left by 12 positions) of the memory address of the frame which is currently storing the contents of the page, if any; since each page is aligned to multiples of 4 KB, the last 12 bits of the address are in fact always set to zero and therefore not stored.

The format of each page directory entry is nearly identical to the one of the page table entries, the only difference being the absence of some bits, replaced by ignored bits. The address of the page table is again stored only for its upper part (20 bits), shifted to the right by 12 bits; this strategy is used also for storing the base address of the current page directory in the CR3 register. The format of the CR3 register and of the paging-structure entries is summarized in the figure below:

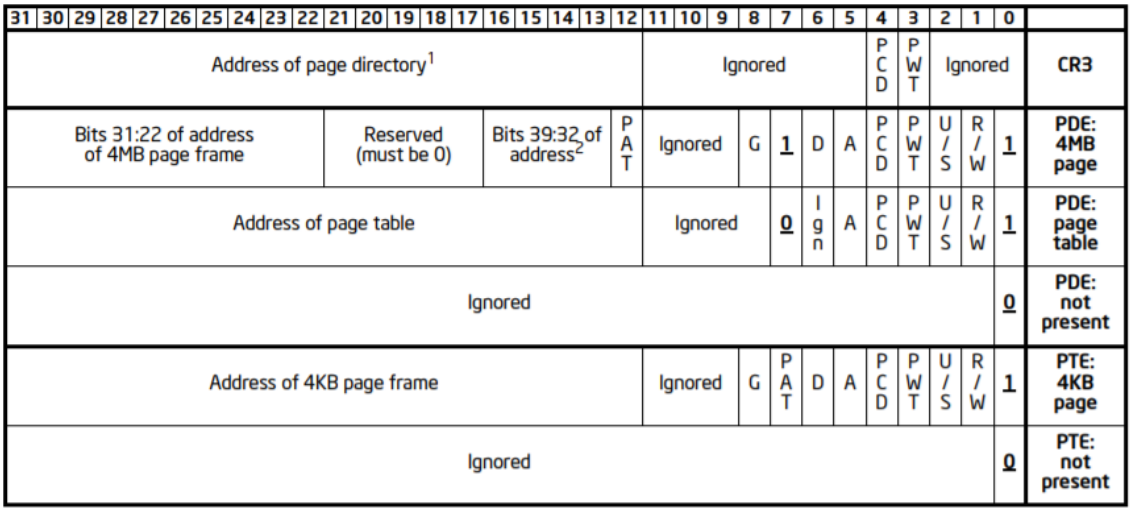

Figure 4-4. Formats of CR3 and Paging-Structure Entries with 32-Bit Paging

In the minimal kernel, each page table entry is represented by a structure of type `page_t`, defined as follows (in the header fine `paging.h`):

```
typedef struct page
{
    u32int present    : 1;
    u32int rw         : 1;
    u32int user       : 1;
    u32int accessed   : 1;
    u32int dirty      : 1;
    u32int unused     : 7;
    u32int frame      : 20;
} page_t;
```

As it can be seen from the definition of the structure, components are stored as bit fields inside a 32-bits long integer, so that in total exactly 32 bits are used for storing each element of type `page_t`; moreover, some of

the fields present in the structure of the page table entry are not reflected in the definition of the `page_t` structure and grouped in the `unused` field (7-bits long). A `page_table_t` type is also defined, as a structure having as its only field an array of 1024 `page_t` elements:

```
typedef struct page_table
{
    page_t pages[1024];
} page_table_t;
```

In order to represent page directories, instead, the `page_directory_t` type is used, defined as follows:

```
typedef struct page_directory
{
    page_table_t *tables[1024];
    u32int tablesPhysical[1024];
    u32int physicalAddr;
} page_directory_t;
```

This type of structure holds:

- An array of pointers to 1024 different page tables;
- An array of 1024 integers, one for each page table, containing the physical address of each of them, for loading it into the CR3 register;
- An integer variable, containing the physical address to the `tablePhysical` array.

These structures were exploited for the solution of the exercise, which is divided in two parts:

- In the first part, it is asked to initialize all the pages the kernel has allocated with an initial value given by the page number;
- In the second part, it is asked to swap the frames corresponding to pages 0 and 1.

For solving the first request of the exercise, an iterative procedure has been implemented, based on the usage of a `for` loop. The loop initializes two variables, `i` and `ptr`, performs the operation `*ptr = i` (copy to the location referenced by `ptr` the value of `i`), prints the contents of the page addressed by `ptr` by means of the implemented `page_print()` function and updates the two variables by incrementing `i` and by adding `PAG_SIZ >> 2` (`0x400`) to the value of `ptr`.

Ideally, in this way we should be able to write the page index on each of the 264 pages the kernel has allocated; in reality, from page 160 onwards (address `0xA0000`), the results are mixed: the operation succeeds in some cases, it fails in some others (the value read from the page is different from the page index) and, in some cases, it even makes the emulated machine restarts (particularly when we write to pages near the end of the allocated memory). This may be due to the fact that addresses in the range `0xA0000` to `0xFFFFF` are reserved, in the x86 architecture, for different purposes (e.g. video memory) and therefore they should not be used as general-purpose memory locations. To avoid any kind of problem, therefore, the `for` loop is stopped when page 160 is reached, meaning when the counter of the pages reaches the value of `PAGE_LIM` (set to 160).

The solution for the second part of the exercise is instead based on the usage of a function called `page_swap()`, defined in `paging.c` and exported in `paging.h` so that it can be called from the main function, which takes as an argument the index of the two pages (each of which is 20 bytes long) whose corresponding frames should be swapped. This function performs three actions:

- It temporarily disables paging, by clearing the uppermost bit of register CR0; this is obtained by reading the value of the register to a `cr0` variable by means of an Assembly `mov` operation, by

performing a bitwise AND operation with the correct mask (logical NOT of `0x80000000`) and by writing back the value of `cr0` again by means of a `mov` operation;

- It swaps the frame address of the two pages, by accessing the page directory at the index given by `page# / 1024` (operation which effectively selects the first 10 bits of the page index) and the page table at the index given by `page# % 1024` (operation which effectively selects the last 10 bits of the page index);
- It re-enables paging by performing the inverse operations it performed in the first step (copy the value of the CR0 register to `cr0`, perform a bitwise OR with `0x80000000`, write back the new value to the register).