Address Translation in the Intel Architecture

1. Segments

Addresses which are output by the 'core CPU' in an Intel Architecture machine are *virtual* addresses in *segmented* form. These addresses are first fed to the Segmentation Unit (SU), which is separate from the CPU core but is internal to the processor chip (it is part of the circuitry called the "Address Translation Unit"). The SU translates the segmented address into a "linear" form – that is, an equivalent address which references (virtual) memory as a contiguous sequence of bytes with linearly increasing addresses.

The address translation (from segmented to linear form) performed by the SU is controlled by a set of "Segment Descriptors" contained in a "Descriptor Table" (either the "Global Descriptor Table (GDT)" or the "Local Descriptor Table (LDT)") in main memory. There are many different Segment Descriptors – one for code references, one for data references, another for stack references, and so forth. These Segment Descriptors must be set up correctly in order for address translation to work properly.

The FLAMES startup code creates a set of Segment Descriptors in main memory – one for code (called the "Kernel Code Segment"), one for data (the "Kernel Data Segment") and so forth. The startup code arranges that when a downloaded program starts running, the CPU can correctly access these Segment Descriptors, and that the values in the Segment Descriptors are such that (virtual) memory appears to the Segmentation Unit translation hardware as one contiguous sequence of bytes – i.e., that every segment, regardless of type, has a starting (virtual) address of "zero" and is 4Gbytes long.

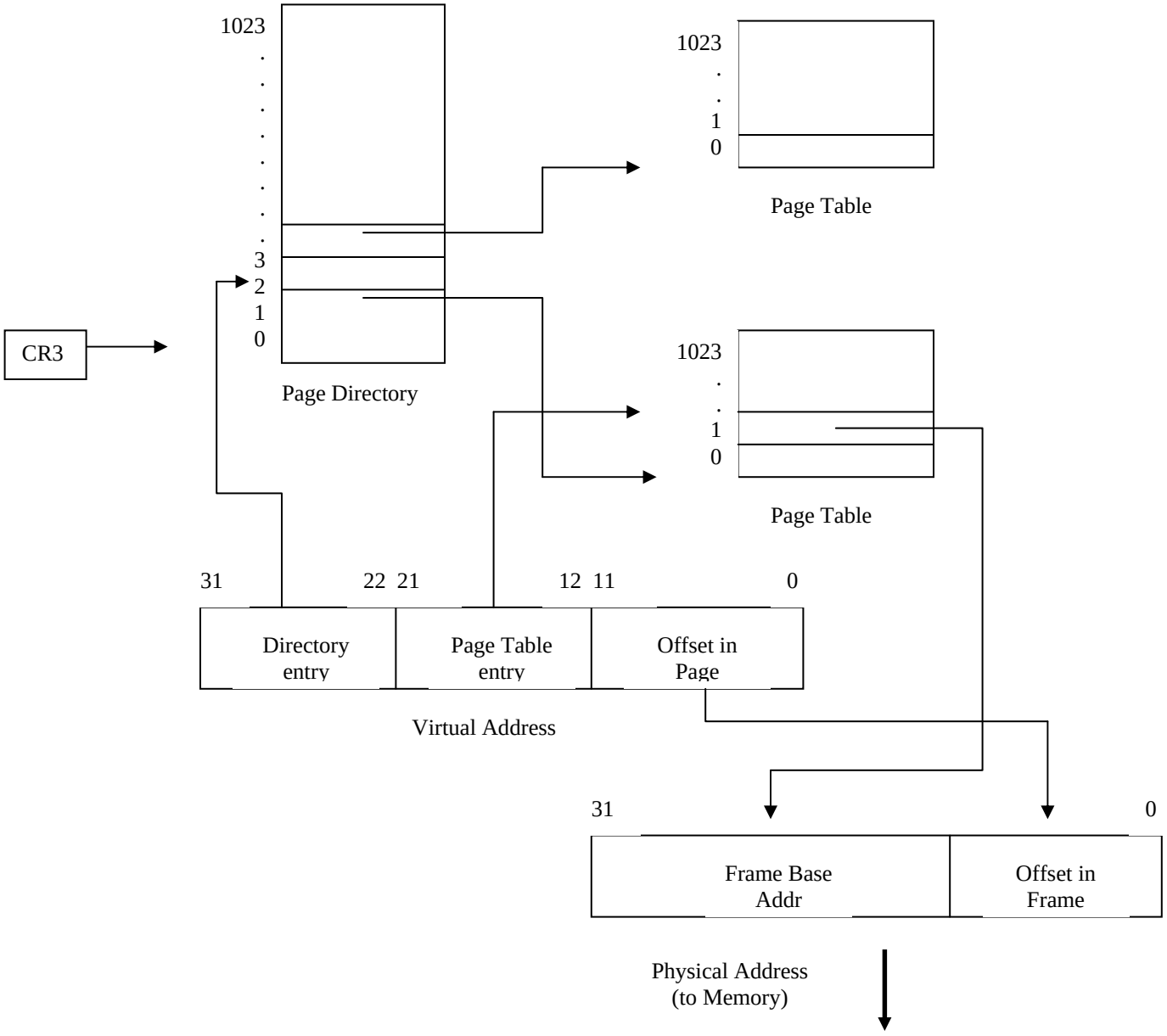
2. Page Translation

Linear (virtual) addresses which are output by the Segmentation Unit are fed into the "Paging Unit". This piece of hardware uses a translation table to perform a mapping from a given virtual address to the corresponding physical memory address. If the mapping is able to be successfully completed, the translated address is output to physical memory (RAM). If not, the Paging Unit generates a *Page Fault* interrupt (INT 14) instead.

The translation table is a hierarchical arrangement of translation values. At the top of the hierarchy is the *Page Directory* table. Entries in the Page Directory table point to *Page Tables*, each of which contains a translation value for a collection of individual virtual space pages. Both the Page Directory table and the individual Page Tables are stored in main memory; CPU Control Register CR3 contains the address of the base of the Page Directory table.

The Page Directory Table contains 1024 entries, each of which is 4 bytes. Each entry contains a pointer to the base of a Page Table, along with some attribute bits. Each Page Table, in turn, contains 1024 4-byte entries, each of which contains a translation value (frame address) of one 4K page of virtual space (again along with some attribute bits). It is these numbers – 1K of Page Directory entries × 1K of Page Table entries × 4K pages – which produce the fact that the size of Virtual Space is 4GB.

The following diagram shows the arrangement of the Page Directory and Page Tables and how a virtual address is translated by the Paging Unit into a physical address.



The Page Directory entries and the Page Table entries have nearly identical formats; they differ in only two bit positions. Page Directory entries have the following structure:

Bits	Meaning
31-12	Upper 20 bits of base address of Page Table
11-9	Available for OS use
8	Global Page (ignored)
7	Page Size (0 = 4K)
6	Reserved (0)
5	Accessed (1 = this Page Table has been accessed)
4	Cache Disabled (1)
3	Cache Policy (0= WriteThrough; 1=WriteBack)
2	User/Supervisor (0 = Supervisor; Page Table cannot be accessed in CPL3)
1	Read/Write (0 = Page Table is Read-Only)
0	Present (1=present)

Page Table entries have the following structure:

Bits	Meaning
31-12	Upper 20 bits of base address of Page (i.e., Frame Number)
11-9	Available for OS use
8	Global Page (ignored)
7	Page Size (0)
6	Dirty (1 = Page has been modified)
5	Accessed (1 = Page has been accessed)
4	Cache Disabled (1)
3	Cache Policy (0= WriteThrough; 1=WriteBack)
2	User/Supervisor (0 = Supervisor; Page cannot be accessed in CPL 3)
1	Read/Write (0 = Page is Read-Only)
0	Present (1 = present)

The logical initial value for a Page Table entry, or a Page Directory entry, for an object which is present in memory is "base_addr + 0x7" – this indicates a page which is user-mode accessible, writeable, and present. The FLAMES startup code creates an initial Page Directory with these values, then allocates a Page Table for each 4Mbyte block of installed physical memory and sets the Base Address value for each Page Table entry to be exactly the same as the corresponding physical frame base address – i.e., it creates what is called "straight-through" or "unity" mapping: every page maps to the frame of the same address. This is the default condition in effect when a downloaded program starts running.

3. Multiple Virtual Spaces

As long as the program which is running uses the default Page Mapping and Segment Descriptors created by the FLAMES startup code, its addresses will be correctly translated from segmented form into linear (virtual address) form, and then translated from virtual to physical form using "straight-through" mapping. Said another way, the running program (the OS Kernel and all its static processes, for example) are running in the same virtual space, and addresses in that virtual space correspond to the same physical memory addresses.

If the Kernel decides to start a piece of code (such as a user process) running in a *different* virtual space, it is necessary to arrange that the new virtual space has a valid set of Segment Descriptors.

The default Segment Descriptors are stored in memory at a place which is accessed through the default Page Table setup. The easiest way to insure the existence of a valid set of Segment Descriptors in a new virtual space is therefore to *copy the Page Table* which is used by the initial (Kernel) code and use the copied Page Table – which points to the default Segment Descriptor Tables which are already built. In other words, a portion of the memory of the existing program (virtual) space – the virtual space shared by the Kernel and the static OS processes which the Kernel creates – is mapped into the virtual space of the new process. The new process then automatically has access to a valid set of Segment Descriptors, and no additional action need be taken to build new descriptors.

One side effect of the scheme of sharing the existing Segment Descriptors by copying the Page Tables is that the "OS code" – the Kernel and the static OS processes – occupy a portion of the virtual space of every user process. This is at least somewhat necessary; at least the Interrupt Descriptor Table must appear in the virtual space of the process – otherwise the process would not be able to invoke Traps successfully.

Once a copy of the default Page Tables is created, it is up to the OS (e.g. the Memory Manager process) to arrange that the Page Table entries which point to the memory locations for the new process code are correctly updated.