10. Virtual Memory

In a typical memory hierarchy for a compute there are three levels: the cache, the main memory and the external storage, usually the disk. So far we have discussed about the first two levels of the hierarchy. the cache and the main memory; as we saw the cache contains and provides fast access to those parts of the program that are most recently used. The main memory is for the disk, what the cache is for the main memory: the programmer has the impression of a very fast, and very large memory without caring about the details of transfers between the two levels of the hierarchy.

The fundamental assumption of **virtual memory** is that programs do not have to entirely reside in main memory when executed, the same way a program does not have to entirely fit in a cache, in order to run.

The concept of virtual memory has emerged primarily as a way to relieve programmer from the burden of accommodating a large program in a small main memory: in the early days the programmer had to divide the program into pieces and try to identify those parts that were mutually exclusive, i.e. they had not to be present at the same moment in memory; these pieces of code, called *overlays*, had to be loaded in the memory under the user program control, making sure at any given time the program does not try to access an overlay not in memory.

Virtual memory, automatically manages the transfer of pieces of code/

data between the disk and main memory and conversely.

At any moment of time, several programs (processes) are running in a computer; however it is not known at compile time, which will be the other programs with which the compiled program will be running in a computer. Of course we discuss here about systems that allow multiprogramming, which is the case with most of the today's computers: a good image for this is any IBM-PC running Windows, where several applications may be simultaneous active. Because several programs must share, together with the operating system, the same physical memory, we must somehow ensure **protection**, that is we must make sure each process is working only in its own address space, even if they share the same physical memory.

Virtual memory also provides **relocation** of programs; this means that a program can be loaded anywhere in the main memory. Moreover, because each program is allocated a number of blocks in the memory, called pages, the program has not to fit in a single contiguous area of the main memory. Usually virtual memory reduces also the time to start a program, as not all code and data has to be present in the memory to start; after the minimum amount has been brought into main memory, the program may start.

10.1 Some Definitions

The basic concepts of a memory hierarchy apply for the virtual memory (the main memory secondary storage levels of the hierarchy). For historical reasons the names we are using, when discussing about virtual memory, are different:

- **page** or **segment** are the terms used for block;
- **page fault** is the term used for a miss.

The name *page* is used for fixed size blocks, while the name *segment* is used for variable size blocks.

	Minimum	Maximum
Page size	512 Bytes	16 Kbytes
Segment size	1 Byte	64 KB - 4 MB

Until now we have not been concerned about addresses: we did not make any distinction between an addressed item and a physical location in memory. When dealing with virtual memory we must make a clear distinction between the address space of a system (the range of addresses the architecture allows), and the physical memory location in a given hardware configuration:

- the **virtual address** is the address the CPU issues;
- the **physical address** results from translating the virtual address (using a hardware and software combination), and can be used to access the main memory.

The process of translating virtual addresses into physical addresses (also named real addresses) is called *memory mapping*, or *address translation*.

The virtual address can be viewed as having two fields, the virtual page/segment number and the offset inside the page/segment. In the case the system uses a paged virtual memory things are simple, there a fixed size field corresponding to offset and the virtual page number is also of fixed size; with a larger page size the offset field gets larger but, in the mean time fewer pages will fit in the virtual address space, and the page number field gets smaller: the sum of sizes of the two fields is constant, the size of an address issued by the CPU.

Virtual page number	Page offset
MSB	LSB

Obviously the number of virtual pages has not to be the same with the number of pages that fit in the main memory, and, as a matter of fact, usually there much more virtual pages than physical ones.

Example 10.1 VIRTUAL PAGE AND OFFSET:

A 32 bit address system, uses a paged virtual memory; the page size is 2 KBytes. What is the virtual page and the offset in the page for the virtual address 0x00030f40?

Answer:
For a page size of N Bytes the number of bits in the offset field is $\log_2 N$. In the case of a 2 KBytes page there are:

$\log_2 2^{11} = 11$ bits

Therefore the number of bits for the page number is:

$32 - 11 = 21$

which means a total of $2^{21} = 2$ Mpages. The binary representation of the address is:

0000 0000 0000 0011 0000 1	111 0100 0000
31 11	10 0

The given virtual address identifies the virtual page number 0x61 = 97_{10}; the offset inside the page is 0x740 = 1856_{10}.

When the virtual memory is segmented, the two fields of the virtual address have variable length; as a result the CPU must issue two words for a complete address, one which specifies the segment, and the other one specifying the offset inside the page. It is therefore important to know from an early stage of the design if the virtual memory will be paged or segmented as this affects the CPU design. A paged virtual memory is also simpler for the compiler.

Another drawback of segmented virtual memory is related to the replacement problem: when a segment has to be brought into main memory the operating system must find an unused, contiguous area in memory where the segment fits. This is really hard as compared with the paged approach: all pages have the same size, and it is trivial to find space for the page being brought.

The main differences between the cache-main_memory and the main_memory_disk can be resumed as follows:

- caches are hardware managed while the virtual memory is software managed (by the operating system). As the example 7.5 points out, the involvement of the operating system in the replacement decision, adds very little to the huge disk access time.

- the size of the cache is independent of the address size of the architecture, while for the virtual memory it is precisely the size of the address which determines the its size.

- besides its role as the bottom level of the memory hierarchy, the disk also holds the file system of the computer.

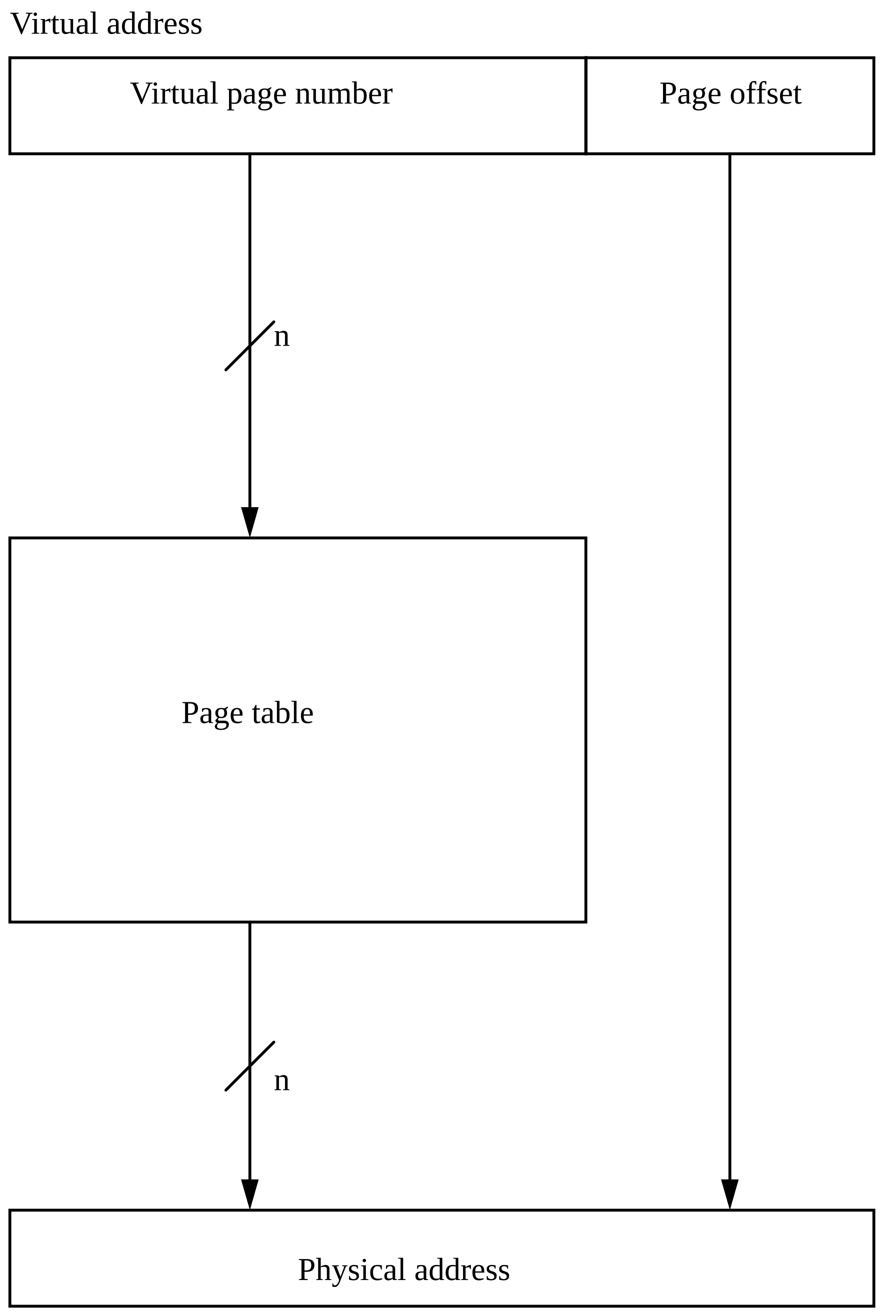

FIGURE 10.1 For a paged virtual memory the page’s physical address (at outputs of page table) is concatenated with the page offset to form the physical address to the main memory.

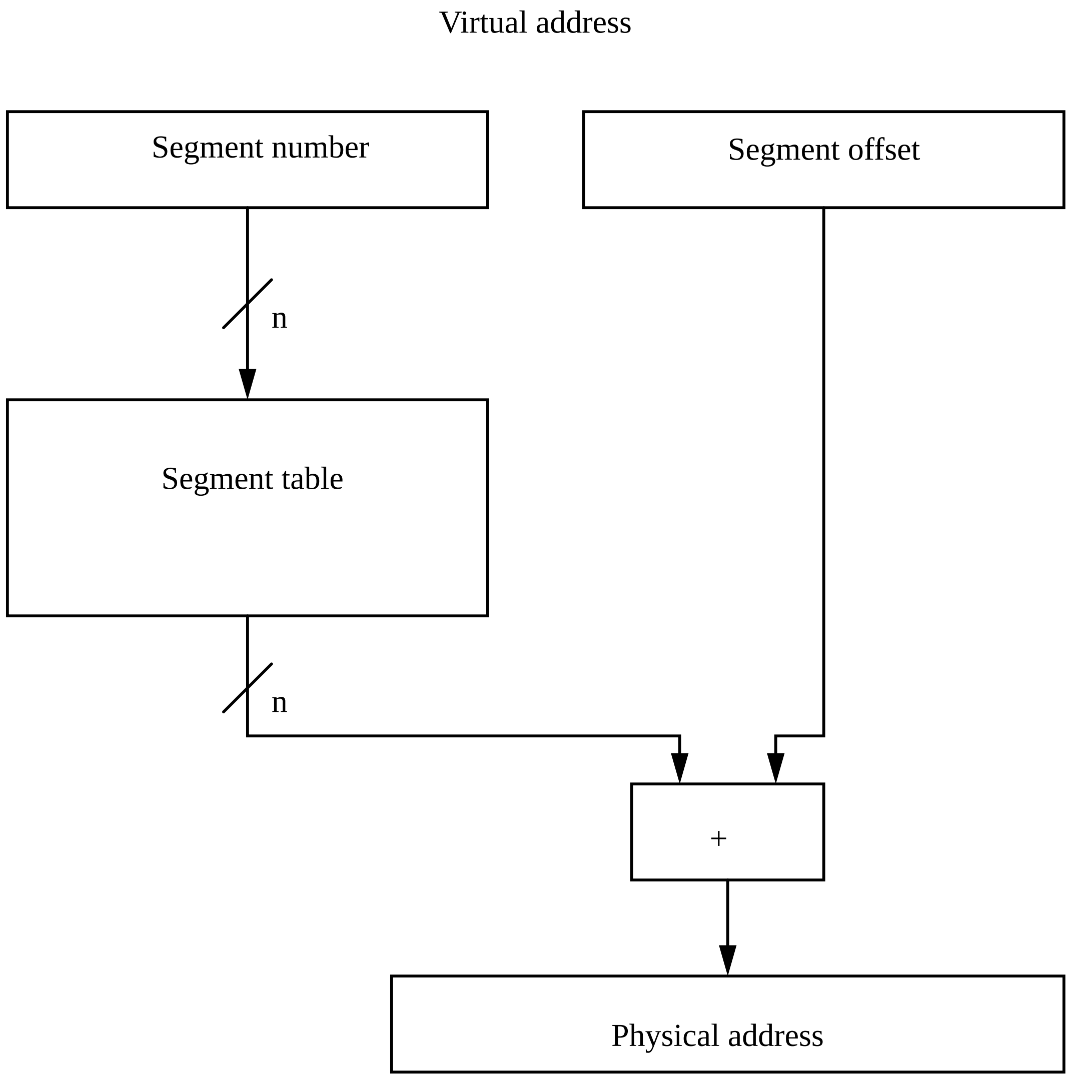

FIGURE 10.2 For a segmented virtual memory the offset is added to the segment's physical address to form the address to the main memory (the physical address).

10.2 How Virtual Memory Works

As with any memory hierarchy, the same basic questions have to be answered.

Where are blocks placed in main memory

Due to the huge miss penalty, the design of the virtual memory focuses in reducing the miss rate. The block placement is a good candidate for optimization. As we saw when discussing about caches, an important source for misses are the conflicts in direct mapped and set associative caches; a fully associative cache eliminates these conflicts. While fully associative caches are very expensive and slow down the clock rate, we can implement a fully associative mapping for the virtual memory. That means any virtual page can be placed anywhere in the physical memory. This is possible because it is in software where misses are handled.

Is the block in the memory or not?

A table ensures the address translation. This table is indexed with the virtual page/segment number to find the start of the physical page/segment in main memory. There is a difference between paging and segmentation:

- paging: the full address is obtained by concatenating the page's physical address and the page offset, as in Figure 10.1

- segmentation: the full address is obtained by adding the segment offset to the segment's physical address, as in Figure 10.2

Each program has its own **page table**, which maps virtual page numbers to physical addresses (a similar discussion can be made about segmentation, but we shall concentrate on paging). This table resides in main memory, and the hardware must provide a register whose content points to the start of the page table: this is the **page table register**, and in our machine can be one of the registers in the general purpose set, restricted to be used only for this purpose.

As it was the case with caches, each entry in the page table contains a bit which indicated if the page corresponding to that entry is in the main memory or not. This bit, the valid bit is set to Valid (1) whenever a page is being brought into the main memory, and to Non-valid (0) whenever a page is replaced in main memory.

Example 10.2 PAGE TABLE SIZE:

A 32 bit address system, has a 4 MBytes main memory. The page size is 1 KByte. What is the size of the page table?

Answer:
The page table is addressed with the page number field in the virtual address. Therefore the number of lines in the page table equals the number of virtual pages. The number of virtual pages is:

$$virtual\_pages = \frac{address\_space\ [\ Bytes\ ]}{page\_size\ [\ Bytes\ ]}$$

$$virtual\_pages = \frac{2^{32}}{2^{10}} = 2^{22} = 4\ Mpages$$

The width of each line in the page table equals the width of the page number field in the virtual address. If the number of virtual pages is 2^{22}, then the line width is 22 bits. It results that the size of the page table is:

page_table_size = number_of_entries * line_size

page_table_size = 2^{22}*22 bits = 11.5 Mbyte!!

As the above example points out the size of the page table can be impressive, in so much that in the example, it does not fit in the memory. However, most of the entries in the page table have the valid bit equal to 0 (Non-valid); only a number of entries equal to the number of pages that fit in the main memory have the valid bit equal to 1; that is only:

$$\frac{main\_memory\_size}{page\_size} = \frac{2^{22}}{2^{10}} = 2^{12} = 4096\ entries$$

have the valid bit equal to 1; this represents only from the total number of entries in the page table.

- One way to reduce the page table size is to apply a hash function to the virtual address, such that the page table must have only so many entries as pages in the main memory, 4096 for the example 10.2. In this case we have an **inverted page table**, The drawback is a more involved access, and some extra hardware.

- Another way to keep a smaller page table is to start with a small page table and a **limit register**; if the virtual page number is larger than the content of the limit register, then the page table must be increased, as the program requires more space. The underlying explanation for this optimization is the principle of locality: the address space won't be uniformly accesses, instead, addresses concentrate in some areas of the total address space.

- Finally, it is possible to keep the page table paged by itself, using the same idea that lags behind the virtual memory. namely that only some parts of it will be needed at any time in the execution of a program.

Even if the system uses an inverted page table, its size is so large that it must be kept in the main memory. This makes every memory access longer: first the page table must be in the memory to get the physical page address, and only then can the data be accessed. The principle of locality works again and helps improving the performance. When a translation from a virtual page to physical one is made, it is probable that it will be done again in the future, due to spatial and temporal locality inside a page. Therefore the idea is to have a cache for translations, which will make most of the translations very fast; this cache, that most new designs provide, is called **translation lookaside buffer** or **TLB**.

What block should be replaced in the case of a miss?

Both because the main purpose is to reduce the miss rate and because the large times involved in the case of a miss allow a software handling, the most used algorithm for page replacing is **Least Recently Used** (**LRU**), which is based on the assumption that replacing the page that has not been used for the longest time in the past will hurt the least the hit rate, in that probably it won't be needed in the near future as it has not been needed in the past.

To help the operating system take a decision, the page table provides a bit, the **reference bit** also called the **use bit**, which indicates if the page was accessed or not. The best candidate for a replacement is a page that has not been referenced in the past; if all pages in the main memory have been referenced, then the best candidate is one page that has been little used in the past and has not been written, thus saving the time for writing it back on the disk.

How are writes handled?

In the case of caches there are two options to manage the writes: write through and write back, each with its own advantages and disadvantages.

For virtual memory there is only one solution: **write back**. It would be disastrous to try writing on disk at every write; remember that the disk access time is in the millisecond range, which means hinders of thousands of clock cycles.

The same improvement that was possible with caches can be used for virtual memory: each entry in the page table has a bit, the dirty bit, which indicates if the content of the table was modified as a result of a write. When a page is brought in main memory, the dirty bit is set to Non-dirty (0); the first write in that page will change the bit to Dirty (1). When it comes to replacing a page, the operating system will prefer to replace a page that has not been written because this saves the time of writing out a page to the disk, again a milliseconds amount of time.

10.3 More About TLB

The TLB is a cache specially designed for page table mappings. Each entry in the TLB has the two basic fields we found in caches: a tag field which is as wide as the virtual page number, a data field which is the size of the physical page number, and several bits that indicate the status of the TLB line or of the page associated with it; these bits usually are a dirty bit which indicates if the corresponding page has been written, a reference bit which indicates if the corresponding page has been accessed, maybe a write protection bit which indicates if writes are allowed in the corresponding page. Note that these bits correspond to the same bits in the page table.

At every memory reference the TLB is accessed with the virtual page number field of the virtual address. If there is a hit, then the physical address is formed using the data field in the TLB for that entry, concatenated with the page offset. If there is a miss in the TLB, the control must determine if there is a real page fault or a simple TLB miss which can be quickly resolved by updating the TLB entry with the proper information from the page table residing in main memory. In the case of a page fault the CPU gets interrupted, and the interrupt handler will be charged to take the necessary steps to solve the fault.

Figure 10.3 presents the logical steps in addressing the memory; the diagram assumes that the cache is addressed using the physical address. The main implication of this scheme is that both accesses, to the TLB and to the cache, are serialized; the time needed to get an item from the memory, in the best case, is the access time in TLB (assume there is a TLB hit), plus the access time in the cache (assume there is a cache hit).

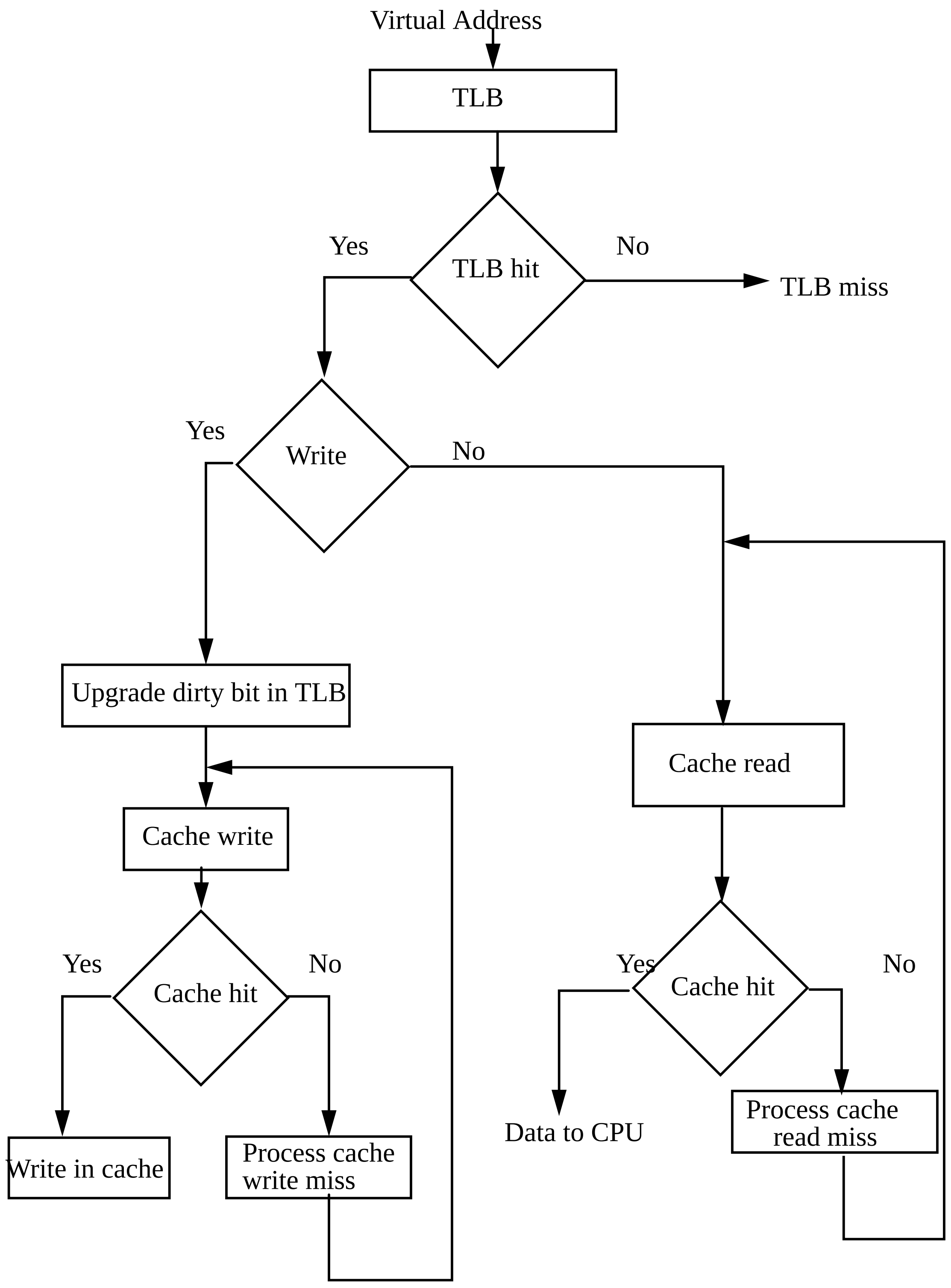

FIGURE 10.3 Steps in read/write a system with virtual memory and cache.

Certainly this is troubling because we would like to get an item from the cache in the same clock cycle the CPU issues the address; with the TLB it is probably impossible to do this, unless the clock cycle is stretched to accommodate the total access time.

Another possibility is the CPU to access the cache using an address that is totally or partial virtual: in this case we have a **virtually addressed cache**. For the virtually addressed caches, there is a benefit, shorter access times, and a big drawback: the possibility of **aliasing**. Aliasing means that an object has more than one name, in this case more than one virtual address for the same object; this can appear when several processes access some common data. In this case an object may be cached in two (or more) locations, each corresponding to a different virtual address. The consequence is that a one process may change data without the other ones being aware of this. Dealing with this problem is a fairly difficult problem.

A final comment about TLB: it is usually implemented as a fully associative cache, thus having a smaller miss rate than other options, with a small number of lines (tens) to reduce the cost of a fully associative cache. For a small number of words even the LRU replacing algorithm becomes tempting and feasible.

10.4 Problems In Selecting a Page Size

There are factors that favor a larger page size and factors that favor a small page size. Here are some. Here are some that favor a large page size:

- the larger the page size is the smaller is the page table, thus saving space in main memory;

- a larger page size makes transfers from/to disk be more efficient, in that given the big access times, it is worth transferring more data once the proper location on the disk has been accessed.

As for factors that favor a small page size, here are some:

- **internal fragmentation**: if the space used by a program is not a multiple of the page size, then there will be wasted space because the unused space in the page(s) can not be used by other programs. If a process has three segments (text, heap, stack), then the wasted space is on the average 1.5 pages. As the page size increases the wasted space gets more relevant

- **wasted bandwidth**: this is related to the internal fragmentation problem. When bringing a page in memory we also transfer the unused space in that page.

- **start time**: with a smaller page many small processes will start faster.

Exercises

10.1 Draw the block schematic of a 32 bit address system, with pages of 4KBytes. The cache has 8192 lines, each 8 words wide. Assume that the cache is word addressable.

10.2 With a TLB and cache there are several possibilities of misses: TLB miss, cache miss, page fault or combinations of these ones. List all possible combinations and indicate in which circumstances they can appear.