1 Class #16

This section examines memory sharing schemes based on mappings. A mapping (MMap) can be defined as a relation from $N- > M$, where N (the domain of the mapping) is the virtual name space that is generated by the processor and M (the range of the mapping) is the phsyical memory address space that the memory system responds to. The Memory Management Unit (MMU) is responsible for translating the virtual addresses generated by the processor into physical addresses that the memory system can respond to.

A simple example of a mapping scheme is the address binding mechanism at run time. In this approach, any address the CPU generates is added to a *base register* value to generate the actual physical address that the memory system responds to.

There are several alternatives for this kind of mapping:

The mapping may be one-one such as in the previous case, where there is a unique physical address corresponding to a given virtual address.

The mapping may be many-to-one. In this case, many virtual addresses can be mapped to one physical address. For example, two different processes may have different virtual addresses that are mapped onto the same physical address. This can happen only when the two processes are *time-multiplexed*, i.e. they occupy the same region of memory at different points of time. The advantage of this picture is that the effective address space available to the system is increased.

In the many-to-one case, since the virtual address space is larger than the physical address space, it is possible in particular that an address generated by the processor does not map to any address in memory. This is in fact the case if some of the data or code of the process is stored on an auxiliary device such as disk and has not been loaded in yet. When this happens, the processor generates a *missing item* fault. When this happens, the running program stops, the operating system intervenes and does the following operations before the running program can resume:

Missing item is located in disk and is transferred to main memory.

If there is no space available in main memory, some item already in the memory is kicked out and the new item is brought in its space.

MMap is altered.

There are several policies that are implicit in this mechanism:

Replacement Rule: Which item is to be kicked out?

Fetch Rule: When should an item be fetched?

Placement Rule: Where should an item be placed, once fetched?

Finally, there is the question of implementation of MMap. This is related to the complexity of the mapping being expressed. The most elaborate mapping scheme would maintain a mapping for each virtual address to its coresponding physical address. However, this mechanism suffers from

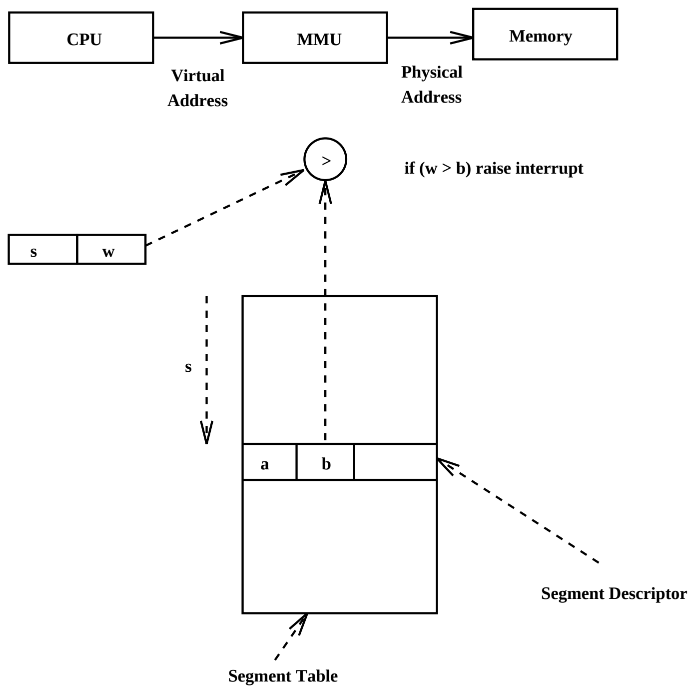

Figure 1: Segmentation

the drawback that the size of the mapping would be basically the size of the memory itself. So, this method is impractical. In practice, a block is chosen as the unit of mapping. This is to say that the start address (virtual) of a block is mapped to a certain physical start address, and a certain block of addresses following the start address are mapped onto address that are consecutive to that physical start address. The special case of block sizes are:

blocksize = memory size. This is when the entire real memory is mapped onto itself, and really no mapping is required.

blocksize = wordsize. This is when each word of virtual address is mapped in a different way to a word of physical address. This scheme is impractical.

Non-trivial block sizes are:

Segmentation Each block (of possibly different size) is mapped in a certain way. Blocks correspond to logical entities, e.g. the entire code of a process may be mapped in one segment and the entire data of a process may be mapped in a certain way.

paging Each block (of the same size) is mapped in a certain way.

Segmentation & paging This corresponds to a hybrid approach, where each segment is divided up into pages.

1.1 Segmentation

Segmentation can be best expressed by Figure 1. Every virtual address the CPU generates consists of two parts: s & w. s is a segment identifier, and is used as an index into a segment table. w

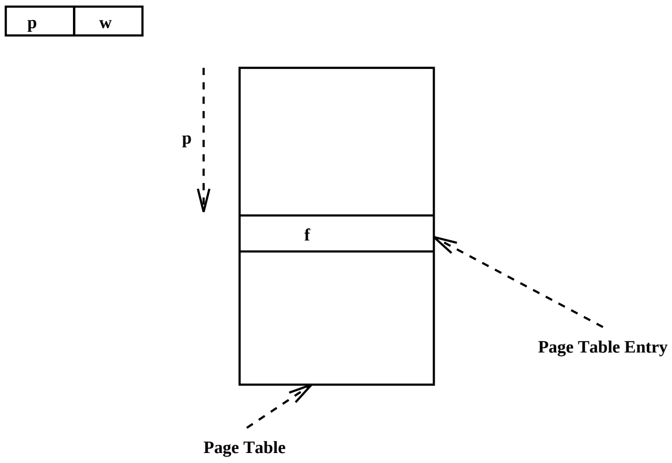

Figure 2: Paging

is an offset within the segment. The segment table is an array of entries, each of which is called a *segment descriptor*. A segment descriptor consists of three fields: a start address field, a length field and a permissions field. If the segment number corresponding to the virtual address points to a segment descriptor with a start address equal to zero, the segment number is invalid. If the segment is valid, then the offset is compared with the length field of the segment. If the offset is out of bounds, then an error is signalled. Access bits are used to determine whether a process can write or read or perform other operations on the memory locations corresponding to the segment. The segment table is typically stored in memory.

A problem with this approach is that every memory access by a process requires two memory accesses. The solution to this is to *cache* the value of all recent translations that have been made from segement identifier to segment start address. This *cache* is typically called the translation lookaside buffer (TLB).

Another problem is that since an entire segment corresponds to a contiguous block of memory, it may not be possible to allocate a segment in memory, simply because there is no single block of memory large enough to allocate a segment. This problem is called **external fragmentation**.

1.2 Paging

Paging is explained by Figure 2. Basically, the physical memory is divided into a series of equal sized *pages*. Virtual addresses are also made up of pages. A virtual page and a physical page have the same number of bytes. A virtual address generated consists of a virtual page number and a page offset. The virtual page number is used as an index into the *page table* to retrive a physical page number. If the virtual page number does not correspond to a currently memory resident page, the page table entry reflects that and an interrupt is raised. If the virtual page number does correspond to a physical page, then the virtual page offset field is appended to the physical page number to get a physical address, which is then addressed by the memory.

Since the number of entries must be equal to the number of virtual pages, the page table can be quite large. For example, with a 32 bit virtual address, and 1024 byte pages, there are as many as 2^{22} entries in the page table, which is quite large. There are several solutions to this problem. A common solution is to use *inverted page tables*. In this mechanism, instead of one entry for every virtual page, there is an entry for every physical page. The entry stores for a given physical page, what is the virtual page that resides there. While this reduces the size of the page table,

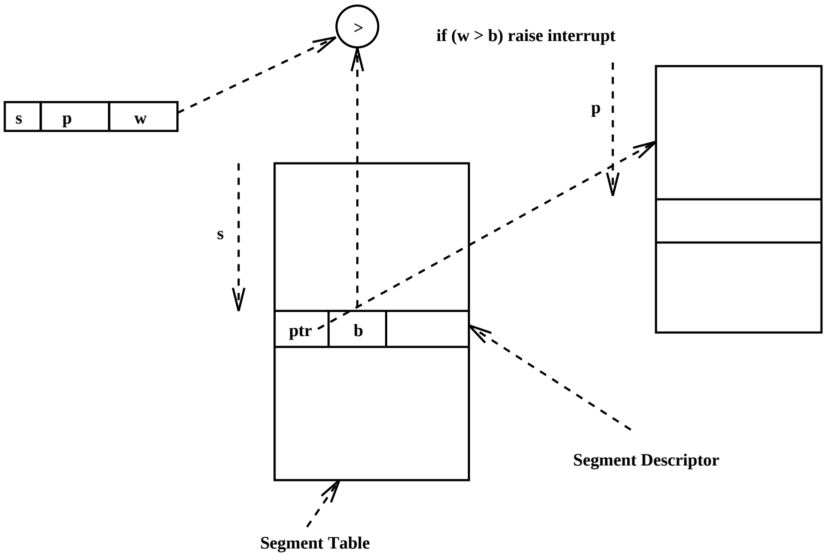

Figure 3: Combining Segmentation and Paging

it comes at the expense of either increased search time, or more pricey hardware(in the form of associative memory). This is because, when a virtual page number is generated, the page table must be searched to see if any physical page table entry matched the virtual page number. Again, TLB caching can reduce this time.

As a final note, it is easy to see that the last page of a process may be left unfilled partially. This problem is known as *internal fragmentation.* On the average, half a page would be wasted per process.

1.3 Combined Approach

As we have seen, the problem with page tables is that the page tables could be quite large. The problem with segments is *external fragmentation.* It is possible to combine both these approaches to alleviate both these problems. This is demonstrated by the scheme in Figure 3.

In this approach, each segment is paged. In other words, instead of the entire segment being contiguous, it is broken up into a set of pages. This solves the problem of external fragmentation. In addition, one page table is maintained for each segment. In other words, the number of entries in the page table are dependent only on the size of the segment, thus ensuring that the size of the page tables does not blow up. The final problem with this approach is that associative memory is a must, because each memory access by the processor now needs *three* memory accesses.