Module 2, Lesson 3 “Paging”

Welcome to this lesson on Paging.

Paging is one way how the memory management unit of the CPU maps the logical address spaces of the various processes to the physical memory.

First we will revisit memory relocation and how it gives rise to external fragmentation.

We will then look at paging, which is basically a way to partition the logical address space into small, equal-sized portions, which we call “pages”, and then relocating these pages individually, thus eliminating external fragmentation.

We will discuss how paging works in details, at least conceptually. For this, we will accompany the memory management unit as it resolves a logical address to get the associated physical address.

We will then notice that unfortunately, paging does not eliminate internal fragmentation.

Let’s begin.

We previously talked about the importance of relocation to map the logical address space onto the physical memory.

Unfortunately, naïve relocation is not particularly useful in practice because it causes external fragmentation.

Let’s illustrate this with an example of a memory management unit that does naïve relocation.

On the left we have our region of logical memory that we want to have mapped to some region in physical memory.

We may have some other regions of memory already mapped to the physical memory, possibly from other address spaces.

Now, in comes a request to map a new region of logical memory, and we need to allocate the necessary physical memory. Because we use naïve relocation, the allocated physical memory has to be contiguous.

This causes a problem in this case, as we don’t have a contiguous portion of physical memory that is large enough to fit the new request. There is enough memory available, just not contiguous. This is a classical case of external fragmentation.

Now, instead of relocating the entire memory region in one piece, we partition it into portions of a pre-defined length, say 4kB.

We call these portions of logical memory “pages”. We do the same for the physical memory, and split it up into portions of the same size, again 4kB each.

We call these fixed-size portions in physical memory “frames”.

Now we relocate each portion in logical memory individually to a portion in physical memory.

We say that we map a page in the logical memory to a frame in physical memory.

We do this for each page of our logical memory region.

We notice that the frames for our logical memory region do not need to be contiguous, or even adjacent. This is because the pages are relocated individually, and independently from each other.

If now we have a new request for memory, say for a memory region in a different address space, the problem of allocating physical memory becomes quite simple.

This region of memory will get partitioned into pages as well, and these pages will be individually relocated, just as we do for our previously allocated region.

As long as we can find available frames, we can allocate the physical memory for any new memory region.

Again, we notice that the frames do not need to be located contiguously or even adjacently.

Paging in Hardware

This relocation of individual pages to frames is called “paging”, and we now look at how paging is realized inside the memory management unit.

While individual relocation units allow for an intuitive depiction of what paging does, this is not how paging is implemented.

So, let’s get rid of them.

Let’s focus instead on how a single logical address is mapped into physical memory.

Let’s go through the steps of how a logical address gets resolved and mapped to the corresponding physical address.

At the beginning the CPU issues a logical address.

This address gets split into two parts.

The first part denoting is the so-called “page number” of the address, which says in which logical page the logical address is.

The second part is the “offset”, which describes which location within the page the address refers to.

The page number is used to index into a so-called “page table”. We can think of the page table as an array that has page numbers as indices.

The entry indexed by the page number contains the number of the frame where the given page is stored in physical memory.

We are now ready to construct the physical address.

For this, we glue together the frame number that we just looked up in the page table, and the offset that we get when we split the logical address. We can use the offset without modification because the offset into the page is the same as the offset into the frame in physical memory.

We can now use the resulting physical address to reference the location in physical memory. This whole operation is called “address resolution”, and is repeated every time the CPU issues a logical address that needs to be resolved in order to access the physical memory at the corresponding physical address.

Paging Address Resolution: Example

Let’s work our way through an example. Let’s assume that we have a PDP-11, which was a popular machine in the seventies and early eighties. The PDP has a 16-bit address space and uses 8kB pages and frames. In order to offset into 8kB, the PDP-11 therefore needs a offset of 13 bits. The remaining 3 bits are used to specify the page. The address space has therefore 2 to the power of 3, that is 8 pages, with 8kB each. Note that this is a unusually small number of pages.

Let’s assume that the CPU issues the address zero in binary.

When we split up this address into page number and offset, the page number is zero, and the offset is zero as well.

We use the page number zero to look up entry zero in the page table, which contains the frame number BINARY 011, that is, decimal 3.

Page zero is therefore stored in Frame number 3.

We concatenate the frame number with the offset from the logical address, and we use the resulting physical address to refer to the physical memory. This physical address refers to the first byte in frame number three in the physical memory.

Paging Address Resolution: Example 2

Let us do another example, using the same PDP-11.

Now we assume that the CPU issues the address BINARY 010 0000000001111.

When we split up this address into page number and offset, the page number is BINARY 010, which is decimal 2, and the offset is decimal 15.

We use the page number decimal 2 to look up the entry number 2 in the page table. This entry happens to contain the frame number BINARY 101, which is decimal 5.

Page 2 is therefore stored in Frame number 5.

We concatenate the frame number with the offset, which is decimal 15, and we use the resulting physical address to refer to the physical memory. This physical address refers to Byte 15 in frame number 5, that is, the sixth frame, in the physical memory.

Internal Fragmentation in Paging

Let's briefly talk about fragmentation in paging memory... As we pointed out earlier, paging successfully eliminates external fragmentation. Let's see how paging does with internal fragmentation.

Let's assume that we have a region of memory that is 13,000 bytes long. If we want to store this region in memory using 4kB sized pages, we need 4 pages.

We store 4kB each in the first three pages, for a total of 12,288 bytes, and the remaining 1,012 bytes in the fourth page.

This means that most of the fourth page, more precisely 3,084 bytes of it, is wasted! This is a classical case of memory wasted to internal fragmentation.

So, we observe that paging suffers from internal fragmentation because we are most likely not filling the last page used to store a memory region.

We may want to estimate how much memory is wasted due to internal fragmentation in paging, and there is really not much intelligent to say other than that in average we waste half of the last frame for each memory region. We therefore waste about half a frame for each allocated region.

This leads to the following question: Should we build our systems to have large frames or small frames? or pages, for that matter.

We trade off fragmentation vs. overhead. Specifically, large frames will lead to more fragmentation, because the half-frame that we waste at the end is now bigger. This becomes expensive whenever we have a large number of relatively small regions.

On the other hand, small frames lead to more overhead. For example, the page table now becomes bigger, and having to store the page table causes more overhead. Also, as we will learn in a following lesson, memory managers often swap pages to secondary storage as part of so-called page faults. Since it is more expensive to ship many small pages back and forth between memory and secondary storage, compared to a few large ones, it is better for frames to be large.

Paging: Summary

Let's briefly recall what we learned in this lesson on paging. In summary, we learned how basic paging works.

In particular we learned that paging helps eliminate external fragmentation, because the memory is partitioned into equal-sized pages, which are individually relocated. When the units to allocate all have the same size, external fragmentation stops being an issue.

We described in detail how the memory management unit maps a logical address to the physical address. This happens by partitioning the logical address into a page number and an offset. The page number is used to look up the frame number in the page table.

Mapping from pages to frames now allows for the logical address space to be much larger than the available physical memory. We note that when a region of the logical address space is not used, there is no need to have a mapping to a frame. The physical memory now only needs to store portions of the logical address space that are actually used.

Finally, we discussed that the fixed page sizes used in paging introduce internal fragmentation. There is a trade-off when picking the size of pages, since small pages lead to a decrease in internal fragmentation, but to an increase in the overhead. The system designer now needs to choose a page or frame size that balances internal fragmentation against performance.

Keep in mind that in this lesson we learned how page tables work conceptually. In the following lessons we will learn how page tables work in practice and how they are implemented.

Thank you for watching!