Paging on the x86

Hello, and welcome to this lesson on Paging on the x86 architecture.

We will be focusing on the mechanics of paging in the 32-bit environment, and ignore paging on 64-bit processors for now.

The x86 memory management unit supports what may be called "paged segmentation", but in practice the segmentation part is largely ignored, and the memory is assumed to be paged using a two-level hierarchical paging mechanism.

Pages and frames on the x86 are 4kB long, as are the page directory and the page table pages. Everything therefore fits into frames. Both the page directory and the page table pages are stored in memory.

Their entries therefore can be manipulated using normal memory read and write operations. We will have a close look at how entries in a page table page look like. We call these page table entries, or PTEs for short.

Page directory entries, or PDEs are structured identically.

We will look at how these entries are used during memory translation. The interesting cases, of course, are when the translation fails, either because the page is missing, or because the memory is write protected, or because of whatever other reason. In such cases a page fault exception is raised, and we will have a closer look at how the CPU supports the handling of the page fault.

The memory management unit on the x86 is quite sophisticated. Among other things, it maintains a lot of metadata that is needed by the operating system to manage the virtual memory.

We will not completely forget the TLB, of course.

It is our hope that this lesson prepares you well for the following programming assignments, where you will be implementing your own physical and then virtual memory manager.

Let's begin.

The x86 has 2-Level Paging

Let's begin by reminding ourselves that the x86 uses a two-level hierarchical paging scheme. As we mentioned earlier, we are ignoring the support for segmentation provided by the x86 MMU.

We have a page table base register, which points to the beginning of the page directory.

And we have a bunch of page table pages, each of which contains a portion of the overall page table.

Let's assume that we want to translate a 32-bit address.

This address gets split up into a 10-bit page table number, a 10-bit page number, and a 12-bit offset.

The page table number is used to index into the page directory to find the page directory entry, PDE in short. The PDE in turn contains the pointer to the beginning of the page table page associated with this page table number.

So now we know which page table page handles our address.

The page number of the address is used to index into the page table page to identify the page table entry, or PTE in short.

The offset is concatenated with the frame number that is stored in the PTE, and the result is the physical address for this references.

Directories and Page Table Pages are in Memory

In reality, of course, the page directory is stored in memory, as are the page table pages.

The page table base register contains the start address of the page directory in memory.

And the page directory entry points to the beginning of the respective page table page in memory.

The memory frame number stored in the page table entry indicates the frame that contains the referenced memory location, and if we concatenate the frame number with the offset we get the physical address.

In the specific case of the x86, the control register 3 is the page table base register. It contains the address of the page directory.

There is another register, control register 0, which has one bit that controls whether paging is turned or not.

If this bit in CR 0 indicates that paging is turned off, then the page table is ignored, and – because we ignore segmentation – the address issued by the CPU does not get transformed. We can think of the CPU referencing physical memory directly.

Whenever the CPU boots up, paging is turned off, and the CPU references physical memory directly, just as illustrated here.

We turn on paging by setting the bit in CR 0.

Starting with the instruction after the bit is set, the MMU starts intercepting and translating references. References therefore don't go directly to physical memory anymore, but rather get translated by the MMU, using the two-step lookup process that we described earlier.

Note that before the MMU can start translating, the page table has to be correctly installed and the entries correctly initialized. Otherwise the system will crash at the next memory reference.

At least, there must be a correctly initialized page directory in memory, with the Page Table Base Register in CR 3 pointing to it. In addition, it will need one or more page table pages that are correctly initialized and that are pointed to by the PDEs in the page directory.

So let's have a closer look at what a Page Table Entry looks like.

Page Table Entries

Let's keep for reference the overview picture of the 2-step translation process in the x86 MMU. We are talking about the Page Table Entry, which is here in this oval in the big picture. What is stored in a PTE?

Well, it is 32-bit long, which would mean that it is well-suited to store the start address of the frame. During the translation process the MMU would simply look up this address and then add the offset to it in order to get the physical address. The problem is that frames all start at frame boundaries, which are at an integer multiple of the frame size, which is 4kB.

The last 12 bits of the frame address are therefore always zero. That would be a waste. Note that we are not drawing to scale here.

So instead of the frame address, we store the frame number, which is 20 bit long. Instead of adding the 12-bit offset to the frame address to get the physical address, we simply concatenate the frame number with the offset. Now we can use the remaining 12 bits to support the CPU and the OS in managing the virtual memory.

Let's look at what the x86 does with these 12 bits.

The least-significant bit is called the "Present" bit, and it indicates whether this entry points to a valid frame. If the portion of the address space represented by this page table entry is not used, there is no need to associate a frame with it, and the "Present" bit in this case would be zero. If the CPU refers to the page

represented by this PTE, a page fault exception occurs. People also like to call this the "valid" bit, because it indicates whether this is a valid page or not. Note that this bit is managed by the operating system.

The next bit indicates whether the page is writeable. If the read/write bit is not set, and the CPU tries to write to the page represented by this PTE, a page fault exception occurs. This bit is managed by the operating system.

The next bit indicates whether this is a user page or a kernel page, and the next two bits are reserved.

The next two bits are managed by the CPU. The accessed, or "use", bit is set by the CPU whenever this page is referenced.

And the "dirty" bit is set by the CPU whenever it write to this page.

The remaining bits are either reserved or available for the operating system to use.

What about Page Directory Entries

Page directory entries in the page directory, are structured identically to page table entries.

The frame number clearly is the number of the frame that contains the page table page.

Some Address Space Arithmetic

Before we dive into page fault mechanics, let's do a bit of address space arithmetic. We have this address that is partitioned into a ten bit page table number, a 10 bit page number, and a 12-bit offset.

Our page table base register points to the page directory.

We use the first 10 bits of the address and index into the directory to get the page directory entry.

Given that we use 10 bits, we can index 2^10, which is 1024 entries. Each entry is 32 bits, which is 4 Byte, long. So we need a total of 4096 Bytes, that 4kB, to store the page directory. Since the frames on the x86 are 4kB each, the page directory fits exactly in a frame.

Each valid PDE points to what we like to call a page table page.

We use the second 10 bits to index into the page table page to get the page table entry.

Again, since we use 10 bits, we can index 1024 different entries, which, at 32 each, need 4kB memory. Thus, the page table page fits exactly in a single frame, just as the page directory.

We use the frame number in the PTE to identify the frame number of the physical location, and we append the remaining 12 bits of the address to complete the physical address to identify the memory location.

These 12 bits naturally address 4kB of memory, which again is the size of a frame. So everything sums up.

We can think of the page directory having 1024 entries, each of which refers to 1024 page table entries, which each manages 4kB. Each page directory entry therefore manages, so-to-say, 4 MB of virtual address space, for a grand total of 4 GB if we count all page directory entries together. This comes to no surprise, given that the 32 bits in the logical address define a 4 GB address space.

Page Fault Exceptions: Mechanics

Let's get back to the page table entry, and how page fault exceptions work. For what we are concerned, the page faults can be triggered by one of three cases.

1. Either the CPU tries to reference an address for which the PTE is invalid.

2. Or the CPU may want to write to a page that is valid, but that is marked as read-only. In this case the Read/Write bit would be zero.
3. Finally, the CPU may be in user mode and the page may be marked as kernel page.

In all cases an Exception 14 is raised. In order to handle this exception, we need to have more information available other than that it happened.

Details about the fault are stored in the error code, which the hardware pushes automatically onto the stack when the exception occurs. As we learned when we discussed exception handling on the x86, the error code can be accessed as part of the register structure by the high-level exception handlers.

This table gives tells us how to interpret the various bits in the error code. We will skip the detailed discussion of the error code because we will be able to make due without it in our projects.

It is also very important to know what address caused the page fault. This address is stored in Control Register 2 of the x86. From this address we can identify the page directory entry, and from that the page table page, and eventually the page table entry. All parts of the PDE and the PTE can be queried using bitwise operations to infer what caused the page fault.

Maintaining Meta-data

Even during successful address translations, the MMU is busy. Among other things it maintains meta-data about the pages to support the virtual memory management of the operating system.

The x86 supports two bits, the accessed, or "use" bit, which is set by the hardware whenever this page is referenced, and the "dirty" bit, which is set by the hardware whenever this page is written to.

The operating system may clear any of these bits and then later check if the page was accessed or written to since the last time the bit was cleared.

We will see how these bits come handy when we discuss paging on demand to and from secondary storage.

Where is the TLB?!

One part of the memory management unit that we have not discussed is the TLB.

This is because the TLB is largely invisible. The TLB cache is loaded transparently, and no user intervention is generally needed.

One case, about which we learned when we discussed TLB in general, is when page table entries are deleted. In such a case we need to explicitly delete the mapping in the TLB as well. This is not done automatically.

The x86 supports a variety of instructions to manage the TLB. The simplest, although by far most inefficient, way to make sure that the TLB is consistent with the page table, is to re-load the page table base register. In the case of the x86, we simply re-load the address of the page table directory into control register CR 2.

Conclusion

We have come to the conclusion of this lesson on Paging on the x86.

We have learned about how the 2-level address translation works, how the page table is laid out in memory, which the page table directory taking one frame, and each page table page one frame as well.

And that manipulations of the page table therefore are done by accessing and modifying page directory entries and page table entries in memory.

We have learned how page faults are handled, and how information is passed to the high-level exception handler.

We have also learned how the hardware maintains useful metadata for the operating system. We will learn in future lessons how this comes in handy when the operating system needs to manage on-demand paging of virtual memory that may be stored in physical memory or on secondary storage devices.

Finally, we have briefly reminded ourselves that this activity is supported by a TLB, and that on the x86 we have to be careful when mappings between virtual and physical memory are deleted. These mappings may have to be deleted in the TLB as well. The easiest, but very inefficient solution is to reload the TLB.

I truly hope that you have enjoyed this lesson on Paging on the x86. With the knowledge gained in this lesson you will be able to implement page fault handlers to manage address spaces for processes and to dynamically manage memory on behalf of processes.

Thank you for watching.