Recursive Page Table Lookup in the x86

Hello and welcome to this brief lesson on **Recursive Table Lookup on the x86** architecture.

We have explored paging on the x86 in a previous lesson, where we learned about the structure of the page directory and the page table pages. We learned that both are stored in memory frames, and that their entries can be easily accessed through their physical addresses.

But when paging is turned on and the CPU starts issuing logical addresses, accessing page table entries gets much more complicated.

In this lesson, we will learn how to access page table entries while paging is turned on. We do this through a technique called Recursive Table Lookup.

Recursive Table Lookup is a trick to access page table directories and page table pages when the CPU has logical address translation turned on.

- **First we will refresh our memory on how the address translation works on the x86.**
- **After that we will illustrate why it is impossible to naively access page directory entries, PDEs, and page table entries, PTEs, through their physical addresses.**
- **Then we will visualize how Recursive Table Lookup can be applied to access page table entries.**
- **After that we will see that the same technique can be used, a bit more aggressively, to access page directory entries as well.**

Operating systems sometimes avoid Recursive Table Lookup by placing page directories and page table pages in portions of the address space that are so-called "**directly mapped**", that is, where the address translation is set up so that the physical address of a location is the same as its logical address. In such cases it makes no difference if we address page directory or page table entries through their physical or through their logical address.

The problem with these approaches is that the directly-mapped area of the address space can easily become full. Moving the tables into the virtual address space of the process allows the system to support many page tables with many page table pages each.

At the end of this lesson you will be able to implement, on the x86, your own page table management where the page directories and page table pages reside in the virtual address space of their processes.

Let's begin.

RECAP: Logical Address Translation in x86

Let's refresh our memory on the logical address translation mechanism on the x86 by walking once through the translation steps.

We have a CPU that is in logical addressing mode and a page table base register that points to the location of the page directory. The directory is stored in a frame in memory.

The CPU now issues a 32-bit logical address, which consists of a 10-bit page-table number – called **X** in this example – a 10-bit page number – called **Y** in this example, and a 12-bit offset, called **D**.

Each time an address is issued, the MMU uses the page table number **X** to index into the page directory. Entry number **X** in the directory, here marked as PDE, contains the frame number of page table page number **X** of the page table.

The **MMU** now uses the second 10 bits of the address, here denoted by "**Y**", to index into this page table page. Entry number **Y** in the page table page, here marked as PTE, contains the number of the frame in memory that contains the referenced memory location.

The MMU uses the remaining 12 bits of the address, here denoted by "**D**", to index into this frame to find the memory location.

We use 12 bits because we reference a single byte. For the PDE and the PTE 10 bits were sufficient because we were addressing 4-byte entries.

The 10 bits for **X** and for **Y** allow to index 1024 different PDEs and PTEs. This is exactly the number of PDEs or PTEs that fit in a frame.

+It is important to remember that this translation is done for each and every address issues by the CPU while paging is turned on.

Think of the CPU continuously repeating the Mantra "**PDE, PTE, Memory Location**", without really caring what the value of the issued address is.

Assessing PTEs and PDEs?!

Let's briefly illustrate the difficulties when we want to modify the page table when logical addressing is turned on.

Say we want to modify this Page Table Entry.

Let’s assume that we know the page table page that contains this entry is located in Frame F_PTP.

The physical address of this entry number **Y** in the page table page is the 20-bit frame number F_PTP, followed by the index **Y**, and by two zero bits. These last two bits are there because entries are 4-byte long, and each starts at a multiple of four byte.

Until now, no problem.

Wrong! There is a problem because the CPU issues logical addresses only!

If we try to access the PTE through its carefully constructed physical address, and the CPU starts going through the address translation using its mantra, everything becomes a big mess, because the index into the page directory is (for all practical purposes) some random number, and the same is the case again for the index into whatever page table page we happen to end up referring to.

Some systems avoid this complication by setting up portions of their address space as “**directly mapped**”, that is, the logical address maps to itself; as a result, the logical address is the same as the physical address. If we place page directory frames and page table pages in directly mapped portions of the address space, entries can be easily accessed using the physical address.

When the number of processes grows, and the memory requirements of the processes grow as well, then the number of frames needed for page-table management may easily exceed the amount of directly-mapped memory. Directories and page table pages are therefore primarily stored in virtual memory.

Addressing a PTE using Recursive Table Lookup

Let’s see how we can access page table entries when the CPU has paging turned on.

Again, we want to access the same PTE, which we mark here in blue.

Its page directory entry has index **X**, and it is the entry number **Y** in this page table page. This is reflected in the address of the memory locations of the page that is managed by this page table entry.

Let’s simplify the picture.

We know that the page directory is stored in Frame F_PD.

One half of the Recursive Table Lookup trick is that we define one entry in the page directory to have a very special meaning.

We pick the last entry, and let the frame number pointed to by this entry not be a page table page, but the page directory itself.

The second half of the Recursive Table Lookup trick is the careful construction of the address to access our blue page table entry.

The structure of this address may make little sense at the beginning, but if you are patient you will discover how this all makes sense in the end.

The beginning of the address is the value 1023, which as we see already, will when used to index into the page directory, will return the funny new entry. (5)

The second part of the address is the value **X**, and the last part of the address is the value **Y**, followed by 2 zeros.

Let's use the MMU mantra to translate this funny address.

We use the first 10 bits to index into the directory and get the PDE. (10)

This gives us the last entry in the directory, which in turn, points to the page table page. By construction, what the MMU thinks is a page table page is actually the page directory again.

Now we use the second 10 bits to index into what the MMU thinks is the page table page.

This gives us page directory entry number **X**, which contains the frame number of the page table page with the blue PTE. Note that the MMU thinks that this page table page is a memory frame.

It therefore uses the last 12 bit to index into the frame, which gives us entry number **Y** in the page table page, which happens to be the blue page table entry. Note that entry number **Y** starts at byte location **Y** times 4 because entries are 4-Byte long.

Addressing a PDE using Recursive Table Lookup

Now that we know how to access page table entries, let's do the same again, this time for page directory entries.

In this example, we want to access the page directory entry No **X**, marked in blue.

We simplify the picture.

And we know the frame number of the directory, call it F_PD.

Same as before, we define the last entry in the directory to contain the frame number of the directory, in this case F_PD.

Similarly to the example before, we construct the address of the blue page table entry to start with 10 bits, which have the value 1023, (5) followed by 10 more bits, with value 1023.

The last 12 bits contain the value **X**, followed by two zeros.

When the MMU translates this address, it strictly follows the mantra, "**PDE, then PTE, then Memory Location",** independent of what the address looks like.

The first 10 bits, with value 1023 to index into the directory. (10)

This returns the last entry in the directory, which contains the frame number of the directory. The MMU now thinks that the directory is the page table page, and proceeds to use the second 10 bits to index into what it thinks is the page table page, but still is the directory.

This returns the last entry in what the MMU thinks is the page table page. The MMU now thinks that the directory is the frame with the requested memory location, which comes in handy because the MMU now uses the last 12 bits to index into the page directory, which it thinks is just some memory frame. At offset **X** times 4 is the blue page directory entry.

Memory Usage of Recursive Table Lookup

One final note about the last entry in the page directory: Because we use the last page directory entry to refer back to the page directory itself, we lose a portion of the address space.

Specifically, we lose 4MB, which is the amount of address space that is managed by one page directory entry.

If we think about it, the size of the entire page table, minus the directory, is 4MB as well.

It also happens to be that we construct the addresses to access the page table entries in a way that the last 4MB of the address space basically map to the page table.

Because 4MB of the address space is missing, the last 4KB of the page table are empty.

This space is not wasted. This is where the page directory is located in the logical address space.

If you don't believe it, go back and check how we construct the address to refer to the page directory entry.

With this we conclude this short lesson on Recursive Table Lookup in the x86.

Thank you for watching.