A Walkthrough of Address Translation

Address translation involves a lot of small details, and figuring out how to trace it out by hand can be a very long and tedious process. Before we start trying to do some translations, make sure you know in general how page tables, virtual addresses, and physical addresses work. Here's a quick review with some important points to remember:

- A virtual address consists of virtual page numbers (or segment numbers – but doing lookups with these is similar to paging so from now on, we'll just reference page tables) and an offset. Virtual page numbers index into page tables, and the offset is used for finding a location inside the physical page.
- We can have multi-level page tables, where the physical page number provided in a page table entry can point to the base of another page table.
- Page table entries contain both a physical page number and some permission bits. Permission bits should be checked at every level of the page table before proceeding.

Now that we have a general idea of how address translation works, here's a more detailed outline of some steps you can follow when tracing through an address translation for some instruction. Note: This is a step-by-step procedure for a trace-through by hand (like what you might find on a midterm), so it doesn't include some things that should happen in a real OS address translation, i.e., page faults, etc.

1. Break the virtual address down into page/segment numbers and offset, based on whatever format your virtual address has.
2. Start from the page table base address, and use the virtual page number as an index to find the page table entry it corresponds to.
3. Retrieve the bytes corresponding to your page table entry.
4. Compare the bytes to the page table entry mapping and check permissions for the instruction you're trying to execute. If the required permissions aren't set, stop immediately.
5. If there is still another level of page table, take the physical page number as the base of the next page table and repeat from step 2. Otherwise, concatenate the physical page number with the original offset to get your completed physical address and perform the instruction.

To illustrate how this works, we'll do an example using Prof. Kubi's Fall 2009 Midterm #1, problem 4, located at http://inst.eecs.berkeley.edu/~cs162/fa10/exams/fa09mt1.pdf.

The first thing to notice here is the layout of the virtual/physical addresses and the page table entries. From these diagrams alone, we can figure out the following information:

- Virtual addresses have two virtual page numbers, which means we have a two-level page table.
- The size of each page table entry is 32 bits = 4 bytes.

From looking at the diagram of memory, we can see that each address points to a byte.

Now let's try doing `Load 0x00003012`.

The bit breakdown of `0x00003012` is `0000 0000 0000 0000 0011 0000 0001 0010`. Comparing this to the virtual address layout, we find that:

- Virtual Page #1 = `0000 0000 00` = index 0
- Virtual Page #2 = `0000 0000 11` = index 3
- Offset = `0000 0001 0010` = `0x12`

(Actually, you can see we didn't have to expand out the offset at all since it doesn't get changed here…but that's a shortcut.) The virtual page numbers here are page table entry indices, not byte addresses! We can multiply these indices by the size of a page table entry to get the offset into the page table, which is where the page table entry we're looking for will be located. From earlier, we know each address points to a byte and a page table entry is 4 bytes, so that means we multiply each page number by 4 to get the address offset:

- Page Table Offset #1 = `0000 0000 00 << 2 = 0000 0000 0000 = 0x000`
- Page Table Offset #2 = `0000 0000 11 << 2 = 0000 0000 1100 = 0x00C`

Now we can start looking up entries. Since it's given that the page table starts at address `0x00200000` and Page Table Offset #1 is `0x000`, this means that our first page table entry is located at address `0x00200000`. Taking the 4 bytes at this location, we get `0x00100007`. From mapping this to the page table entry layout, we find that:

- Physical Page # = `0x00100`
- Permissions set: User, Writable, Valid

At this point, we check permissions first. The ones applicable here are User and Valid. Valid is always checked, and since we are specified as running in user mode, User must also be on. This particular instruction is a load so the Writable permission doesn't really matter, but it would be important if it was a store instead. The permissions here look good, and since we know that there is still one more level of page table, we take the physical page # and concatenate it with Page Table Offset #2 to find the address of our next page table entry. This gives us the address `0x0010000C`, which contains the entry `0x00004007`. Similar to the above, this means:

- Physical Page # = `0x00004`
- Permissions set: User, Writable, Valid

Again, the permissions look good. Now there are no more levels of page table left, so this physical page # can be concatenated with the original page offset from the virtual address to get the physical address we were looking for. This gives us the address `0x00004012`, and from here we do our load instruction to get the byte `0x84`.

Congratulations! You successfully performed an instruction using address translation! Any other instruction will follow these same steps – all you have to do is make sure you know what permissions you need to check for.