8multi-level-paging-lab

CST 334 (Operating Systems)
Dr. Glenn Bruns

Lab: Multi-level paging

Note: this lab takes a long time. Considering editing it by removing the simulator, and moving the simulator work to a homework.

1. Draw a picture showing the parts of a virtual address when multi-level paging is used.

2. Draw a picture showing how a virtual address is translated to a physical address when multi-level paging is used.

3. Suppose we are working with pages that are 256 bytes in size, using multi-level paging, and that our page directory has 32 rows. Suppose also that each row of a page table requires 4 bytes of memory. How many bits in a virtual address?

4. You can find a simulator for multi-level paging here:

/home/CLASSES/brunsglenn/OSTEP/HW-Paging-MultiLevelTranslate

The information in the README file says that we are using 32 byte pages, physical memory has 128 pages, and a virtual address is laid out like this:

xxxxx xxxxx xxxxx

where the 5 bits on the left are an index into the page directory, the 5 middle bits index into a “page table chunk”, and the 5 bits on the right are an offset into a page.

Each row of the **page directory** is 1 byte in size. A row has this format:

xxxxxxxx

The red bit is 1 is the entry is valid. If so, then the blue bits give the PFN of a page table chunk.

Each row of a **page table chunk** is also 1 byte in size. A row has this format:

xxxxxxxx

Again, the red bit is 1 if the virtual page associated with that row is valid. If so, then the blue bits give a PFN.

5. Run the program without any command-line options. Your job is to translate virtual addresses to physical addresses using the page directory. Let's go through the steps:

Where is the page directory? The program output shows the page where the page directory starts is 108 decimal.

Which page directory entry? At the bottom of the output is the base address of the page directory (108 decimal) and then a bunch of virtual addresses. Let's try the first one, for virtual address 611c. In binary, that is:

0110 0001 0001 1100

Breaking this up into the parts of the virtual memory address, we get:

0110 0001 0001 1100 **(virtual address)**

The orange bits (11000 binary, or 24 decimal) are an index into the page directory, which starts at page 108 (decimal).

So we look at page 108, and get the 25th byte on the line, because we want byte number 24, starting at 0. We get value 0xa1.

Which page table chunk? Let's look at the page table entry we just found in binary:

10100001 **(page directory entry)**

The red bit is 1, so the entry is valid.

The rest tells us that the need the page table chunk located at page 33 decimal.

Which entry in the page table chunk? Now we index into that chunk using 01000 (8 decimal), which are the 5 bits in the middle of the virtual address. Looking at item 8 (counting from 0) in page 33, we get value 0xb5.

Which physical page frame? Let's look at the entry we just got from the page table chunk in binary:

10110101

The red bit is 1 -- awesome, the virtual page is valid, so the blue bits are the page frame number, which is 53 decimal.

Which byte in the physical page frame? The 5 right-hand bits in the virtual address are 11100 binary (28 decimal), which is the offset into page frame number 53.

What is the physical address? Putting together the page frame number we found in the page table (0110101), plus the offset in the virtual address (11100), we get:

0110 1011 1100

which is 0x06bc. To get the actual value, we look at the 29th entry in page 53, which is 0x08 (8 decimal).

6. Check the answer by running the program again, but with the -c flag. Now run the program again without the -c flag, and try to translate the other virtual addresses at the bottom of the output.

7. If you still have time, repeat 6 but use a seed value when you call the simulator, so that you get a different set of problems.