TLBs, Large Pages, Boot sector

TLB

- optimization: CPU's TLB caches vpn => ppn mappings
- Typical size: 12-4096 entries
- typical hit time: 0.5-1 clock cycle
- typical miss penalty: memory access (10-100 clock cycles)
- typical miss rate: 0.01-1%
- Typically fully associative (to minimize miss rate)
- Cache replacement policy: e.g., FIFO, LRU, etc.
- It is important for TLB miss rates to be really low (e.g., 0.1-1%) for paging to be a useful idea. Fortunately, this is true in practice because programs typically exhibit large temporal and spatial locality.
- if you change any part of the page table, you must flush the TLB!
  - by re-loading %cr3 (flushes everything). e.g., on context switch
  - by executing `invlpg` *`va`*

  So, neither write back, nor write through. Requires explicit invalidations and flushes. Thus OS needs to be careful, e.g., if it overwrites the page table but does not execute an appropriate `invlpg` instruction, a user process may be able to access another process's page.

Large Pages

- x86 also supports "large pages" of size 4MB. This allows large processes to be mapped with less page table space overhead, less translation time, and most importantly, less TLB pressure. For example, if a 4MB page can be placed contiguously in physical memory, it requires only 1 TLB entry to be mapped. If we had used regular 4KB pages, 1024 entries would have been required.
- With large pages, the top 10 bits of the virtual address (VA) are used to index into the page directory. The top 10 bits of the PDE are used to identify the Physical Page Number (PPN). The last 22 bits of the virtual address are used as an offset into the physical page. The last 22 bits of the PDE are unused (or used for storing flags), when using large pages.
- Depending on the size of the process and its access patterns in memory, the kernel may decide to use large or small pages for it (to optimize performance).
- One immediate use of large pages is to map the kernel's pages. Because the kernel typically maps the entire physical memory as one contiguous chunk in the VA space, large pages can significantly reduce the number of page table entries required to implement this mapping (thus saving physical memory space). More importantly, mapping the kernel in this way reduces TLB pressure and also improves translation time in case of TLB miss.

Boot sector

- Only disk state persists across power cycles (off and on). Thus the kernel code/data live on disk, so they can be loaded on bootup.
- The disk is divided into *sectors* (also called *blocks*) of size 512 bytes each.
- Logically, the disk can be thought of as a linear array of sectors, starting with sector number `0`.
- The first sector (number `0`) is special, and is called the bootsector.
- The hardware (or BIOS) promises to load the bootsector (of 512 bytes) at physical addresses `0x7c00-0x7e00`, and transfers control to address `0x7c00`.
- The OS developer needs to write the contents of the bootsector such that it contains the code instructions and the associated data to load the kernel from the disk into memory and transfer control to it. The code and data to do this, must fit within 512 bytes.
- We will next look at the bootsector code of xv6 for a concrete example.

xv6 Bootsector

- Look at Sheet 84, line 8412 (`bootasm.S`)
  - The first instruction at physical address 0x7c00 is the instruction at this line, namely `cli`.
  - `cli` is used to clear the interrupt flag (IF). The interrupt flag is a bit in the EFLAGS register which indicates if the processor accepts external interrupts or not. If `IF=1`, and an external interrupt occurs (by setting the `INTR` pin in the chip by an external device), the corresponding interrupt handler is invoked by transferring control to it through the IDT. On the other hand, if `IF=0`, any external interrupt is ignored.
  - Before this instruction, the BIOS may have installed some interrupt handlers. The kernel now wants to reinitialize everything, and will install its own interrupt handlers. However, at this point, it has not installed any handlers, so it must disable any external interrupts using the `cli` instruction.
- Look at lines 8415-8418. These instructions load zero into all the relevant segments, i.e. `ds, es, ss`. Recall that the processor is executing in 16-bit mode at this time, and so segmentation works by simply multiplying the segment register by 16 and adding it to the offset to obtain a physical address. By zeroing out the segment registers, the programmer intends to use a flat address space.
- Ignore lines 8422-8436. This code enables the processor to access physical addresses above 2^{20}. Recall that the original 16-bit 8086 processor did not allow physical addresses above 2^{20}, and so this sequence of instructions ensures that this is allowed from now on.
- Line 8441: `lgdt` loads the global descriptor table. At this point, the programmer is preparing to switch to 32-bit mode with segmentation enabled (also called protected mode). The operand of the `lgdt` instruction is a GDT descriptor `gdtdesc`, [see line 8487] which contains

the size of the GDT (first two bytes) and the base address of the GDT (next four bytes).

- The base address of the GDT (as given in `gdtdesc`) is specified by the `gdt` variable, which is defined at lines 8482-8485. These lines specify the first three entries of GDT (the size of the GDT is only 3 entries). The contents of these three entries are constants that refer to the corresponding : the first entry specifies a NULL segment, the second entry specifies a segment that is readable and executable and starts at 0 (base) and ends at 0xffffffff (limit), and the third entry specifies a segment that is writable and also starts at 0 (base) and ends at 0xffffffff (limit).
- Thus, the programmer initializes a flat address space (no segmentation), given that it initializes its base to 0 and limit to 2^{32}-1. (Notice that the macros SEG_NULLASM and SEG_ASM are macros defined elsewhere).
- Also, the programmer will load the first descriptor into CS, and the third descriptor into the other segments (DS, ES, SS).
- Ignore the next three instructions (8442-8444). They set the protected-mode bit in the control register `cr0`.
- Finally, the programmer executes `ljmp` instruction to load the CS and EIP registers. Notice that the CS register is loaded with `SEG_KCODE * 8` - this effectively points it to descriptor number 1 (SEG_KCODE=1), in privileged mode (the last two bits are zero). Also, it loads the EIP with `start32` which is the address of the following instruction.
- The assembly code uses the `.code32` directive to instruct the assembler to assemble all future code for 32-bit mode.
- The first few instructions in 32-bit mode (8458-8461) setup DS, ES, and SS registers to point to the second segment descriptor in privileged mode (SEG_KDATA * 8).
- Also, FS and GS are made to point to the NULL segment descriptor as the programmer is not planning to use these segments at all. (Recall that DS, ES, and SS are default segments for some instructions, so they definitely need to point to a valid segment).
- Line 8467: Here, the stack is initialized to point to `start` (whose value is 0x7c00, as that's where the execution starts). Because the stack grows downwards, the space below 0x7c00 is used for the stack frames. The stack is initialized because in the next instruction, a function call will be made which will push the return address to stack.
- Finally, the assembly makes a function call into the C function, `bootmain`. `bootmain` also points in the bootsector region (0x7c00-0x7e00), just like `start` and `start32`. `bootmain` is written in C and compiled into binary code before storing it in the bootsector.
- Because the stack has been initialized, all local variables of `bootmain` will be allocated on the stack.