Example hardware for address spaces: x86 segments

The operating system can switch the x86 to protected mode, which supports virtual and physical addresses, and allows the O/S to set up address spaces so that user processes can't change them. Translation in protected mode is as follows:

- selector:offset (virtual / logical addr)
  ==SEGMENTATION==>
- linear address
  ==PAGING ==>
- physical address

Six segment registers:

%cs, %ss, %es, %fs, %gs, %ds

%cs – EIP, %ss – ESP, %ds – default data segment, %es – string operation

movsb – mov byte at %ds:(%ESI) to %es:(ESI)

rep movsb – copy ECX bytes from %ESI, to %EDI

If `EFLAGS.DF` is `0`, `ESI` and `EDI` are incremented, otherwise they're decremented.

- segment register holds segment selector
- selector: 13 bits of index, local vs global flag, 2-bit RPL
- selector indexes into global descriptor table (GDT)
- segment descriptor holds 32-bit base, limit, type, protection
- la = va + base ; assert(va < limit);
- choice of seg register usually implicit in instruction
  - ESP uses SS, EIP uses CS, others (mostly) use DS
  - some instructions can take far addresses:
    - `ljmp $selector, $offset`
- GDT lives in memory, CPU's GDTR register points to base of GDT
- LGDT instruction loads GDTR
- you turn on protected mode by setting PE bit in CR0 register
- What about protection?
  - instructions can only r/w/x memory reachable through seg regs
  - not before base, not after limit
  - can my program change a segment register? yes, but... to one of the permitted (accessible) descriptors in the GDT
  - can my program re-load GDTR? no!

    - how does h/w know if user or kernel?
    - Current privilege level (CPL) is in the low 2 bits of CS
    - CPL=0 is privileged O/S, CPL=3 is user
    - why can't app modify the descriptors in the GDT? it's in memory...
    - what about system calls? how do they transfer to kernel?
    - app cannot **just** lower the CPL

Why do we care about x86 address translation?

- It can simplify s/w structure: addresses in one process not constrained by what other processes might be running.
- It can implement tricks like demand paging and copy-on-write.
- It can isolate programs to contain bugs or increase security.
- It can provide efficient sharing between processes.

Why aren't protected-mode segments enough?

- Why did the 386 add translation using page tables as well?
- Isn't it enough to give each process its own segments?
- Programming model, fragmentation
- In practice, segments are little-used

Translation using page tables (on x86):

- segmentation hardware first computes the linear address
- in practice, most segments (e.g. in JOS, Linux) have base 0 and max limit, making the segmentation step a no-op.
- paging hardware then maps linear address (la) to physical address (pa)
- (we will often interchange "linear" and "virtual")
- when paging is enabled, every instruction that accesses memory is subject to translation by paging
- paging idea: break up memory into 4096-byte chunks called pages
- independently control mapping for each page of linear address space
- compare with segmentation (single base + limit): many more degrees of freedom
- 4096-byte pages means there are 2^20 = 1,048,576 pages in 2^32 bytes
- conceptual model: array of 2^20 entries, called a page table, specifying the mapping for each linear page number
- table[20-bit linear page #] => 20-bit phys page #
- PTE entries: bottom of handout
- 20-bit phys page number, present, read/write, user/supervisor, etc
- puzzle: can supervisor read/write user pages?

- can use paging hardware for many purposes
    - (seen some of this two lectures ago)
    - flat memory
    - segment-like protection: contiguous mappings
    - solve fragmentation problems when allocating more memory (xv6-like process memory layout)
    - demand-paging (%cr2 stores faulting address)
    - copy-on-write
    - sharing, direct access to devices (e.g. /dev/fb on linux)
    - switching between processes
- where is this table stored? back in memory.
- in our conceptual model, CPU holds the physical address of the base of this table.
- %cr3 serves this purpose on the x86 (with one more detail below)
- for each memory access, access memory again to look up in table
- why not just have a big array with each page #'s translation?
- same problems that we were trying to solve with paging! (demand-paging, fragmentation)
- so, apply the same trick
    - we broke up our 2^32-byte memory into 4096-byte chunks and represented them in a 2^22-byte (2^20-entry) table
    - now break up the 2^22-byte table into 4096-byte chunks too, and represent them in another 2^12-byte (2^10-entry) table
    - just another level of indirection
    - now all data structures are page-sized
- 386 uses 2-level mapping structure
- one page directory page, with 1024 page directory entries (PDEs)
- up to 1024 page table pages, each with 1024 page table entries (PTEs)
- so la has 10 bits of directory index, 10 bits table index, 12 bits offset
- %cr3 register holds physical address of current page directory
- puzzle: what do PDE read/write and user/supervisor flags mean?
- now, access memory twice more for every memory access: really expensive!
- optimization: CPU's TLB caches vpn => ppn mappings
- if you change any part of the page table, you must flush the TLB!
    - by re-loading %cr3 (flushes everything)
    - by executing `invlpg` *`va`*
- turn on paging by setting CR0_PE bit of %cr0
- Here's how the MMU translates an la to a pa:
- ```
  uint
  translate (uint la, bool user, bool write)
  {
    uint pde;
  ```
  ```

```
      pde = read_mem (%CR3 + 4*(la >> 22));
      access (pde, user, write);
      pte = read_mem ( (pde & 0xfffff000) + 4*((la >> 12) & 0x3ff));
      access (pte, user, write);
      return (pte & 0xfffff000) + (la & 0xfff);
    }

    // check protection. pxe is a pte or pde.
    // user is true if CPL==3
    void
    access (uint pxe, bool user, bool write)
    {
      if (!(pxe & PG_P)
         => page fault -- page not present
      if (!(pxe & PG_U) && user)
         => page fault -- not access for user

      if (write && !(pxe & PG_W)) {
        if (user)
           => page fault -- not writable
        if (%CR0 & CR0_WP)
           => page fault -- not writable
      }
    }
```

Can we use paging to limit what memory an app can read/write?

- user can't modify cr3 (requires privilege)
- is that enough?
- could user modify page tables? after all, they are in memory.

Physical address extensions

Enabling PSE (by setting bit 4 in cr4) changes these scheme. The entries in the page directory have an additional flag in bit 7, named PS (for page size). Page directory entry with PS bit set does not point to a page table but to a single large 4 MiB page.