Address translation and sharing using page tables

Reading: 80386 chapters 5 and 6

Handout: **x86 address translation diagram** - PS - EPS - xfig

Why do we care about x86 address translation?

- It can simplify s/w structure by placing data at fixed known addresses.
- It can implement tricks like demand paging and copy-on-write.
- It can isolate programs to contain bugs.
- It can isolate programs to increase security.
- JOS uses paging a lot, and segments more than you might think.

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?

Translation using page tables on x86:

- paging hardware maps linear address (la) to physical address (pa)
- (we will often interchange "linear" and "virtual")
- page size is 4096 bytes, so there are 1,048,576 pages in 2^32
- why not just have a big array with each page #'s translation?
    - table[20-bit linear page #] => 20-bit phys page #
- 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
- What's in a PDE or PTE?
    - 20-bit phys page number, present, read/write, user/supervisor
- cr3 register holds physical address of current page directory
- puzzle: what do PDE read/write and user/supervisor flags mean?
- puzzle: can supervisor read/write user pages?
- 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, read);
  pte = read_mem ( (pde & 0xfffff000) + 4*((la >> 12) & 0x3ff));
  access (pte, user, read);
  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
    else if (!(pxe & PG_U))
      => page fault -- not writable
    else if (%CR0 & CR0_WP)
      => page fault -- not writable
}
```

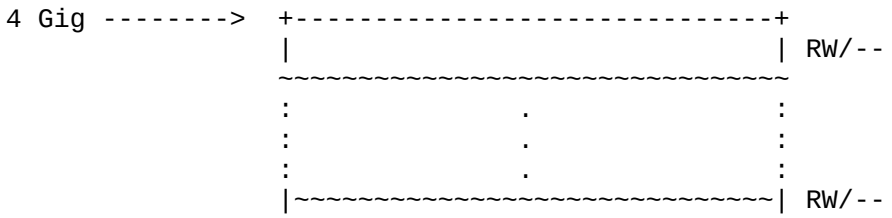
- CPU's TLB caches vpn => ppn mappings
- if you change a PDE or PTE, you must flush the TLB!
    - by re-loading cr3
- turn on paging by setting CR0_PE bit of %cr0

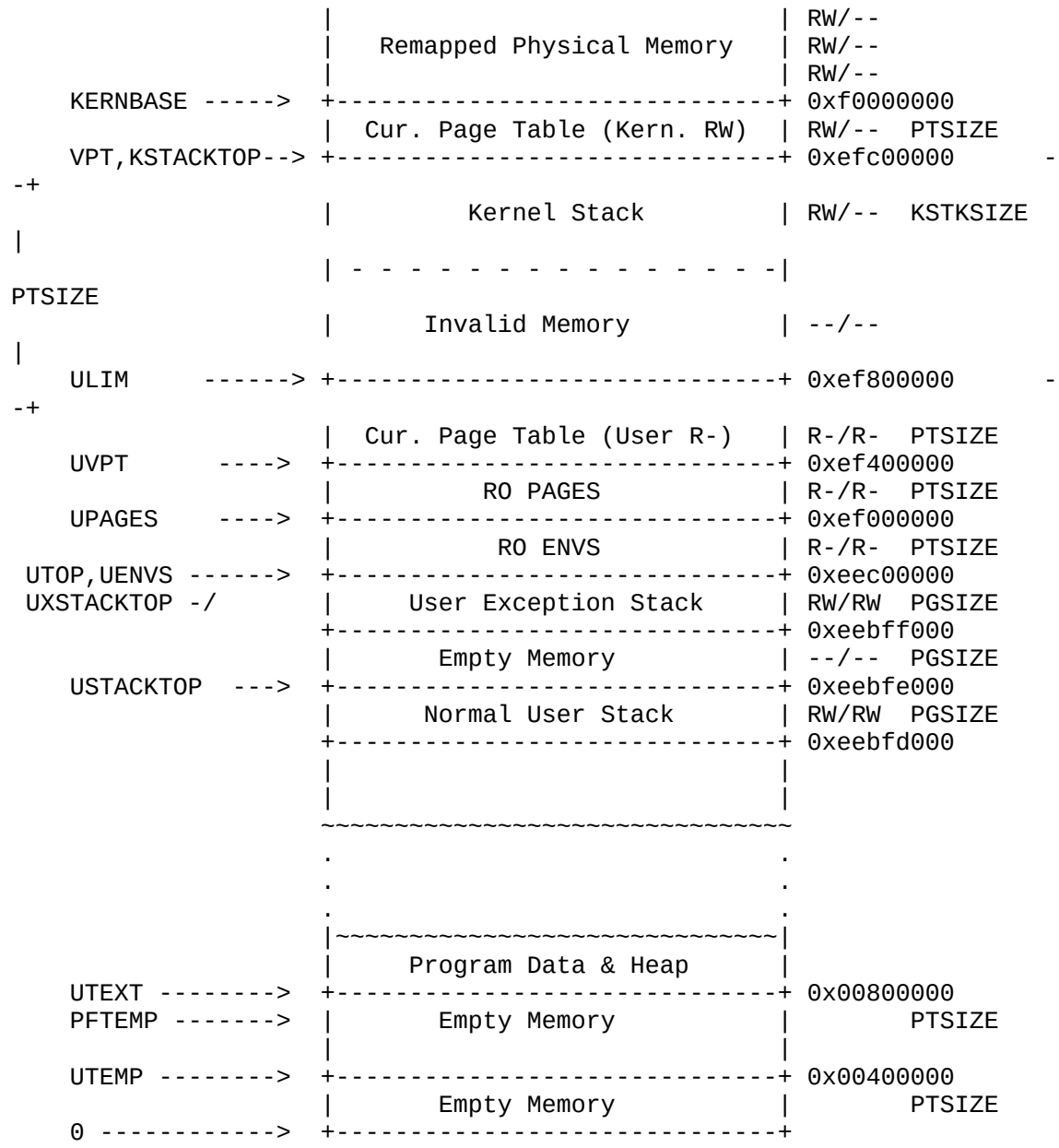
Can we use paging to limit what memory an app can read/write?

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

How we will use paging (and segments) in JOS:

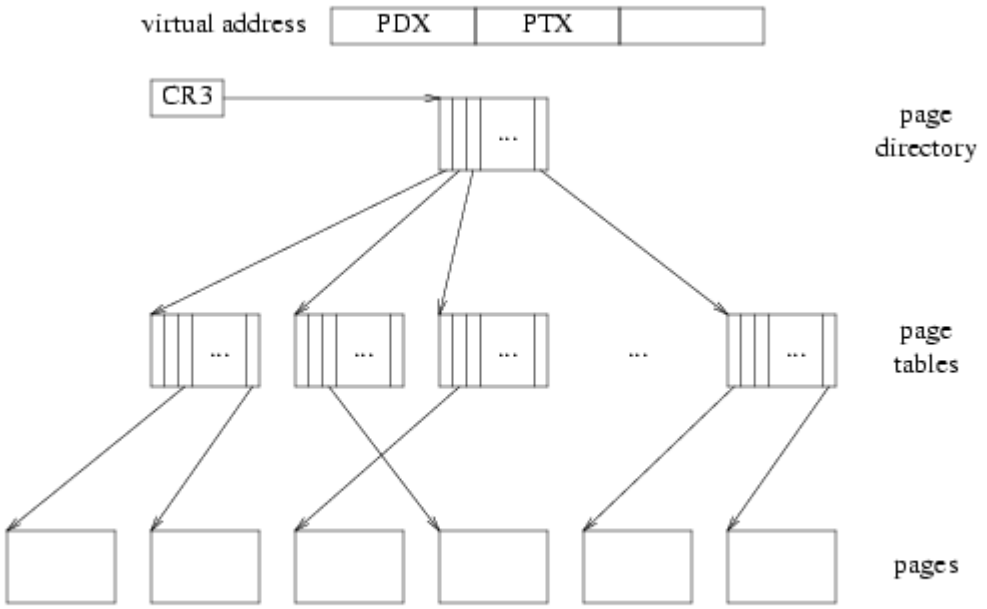
- use segments only to switch privilege level into/out of kernel
- use paging to structure process address space
- use paging to limit process memory access to its own address space
- below is the JOS virtual memory map
- why map both kernel and current process? why not 4GB for each?
- why is the kernel at the top?
- why map all of phys mem at the top? i.e. why multiple mappings?
- why map page table a second time at VPT?
- why map page table a third time at UVPT?
- how do we switch mappings for a different process?





The VPT

Remember how the X86 translates virtual addresses into physical ones:

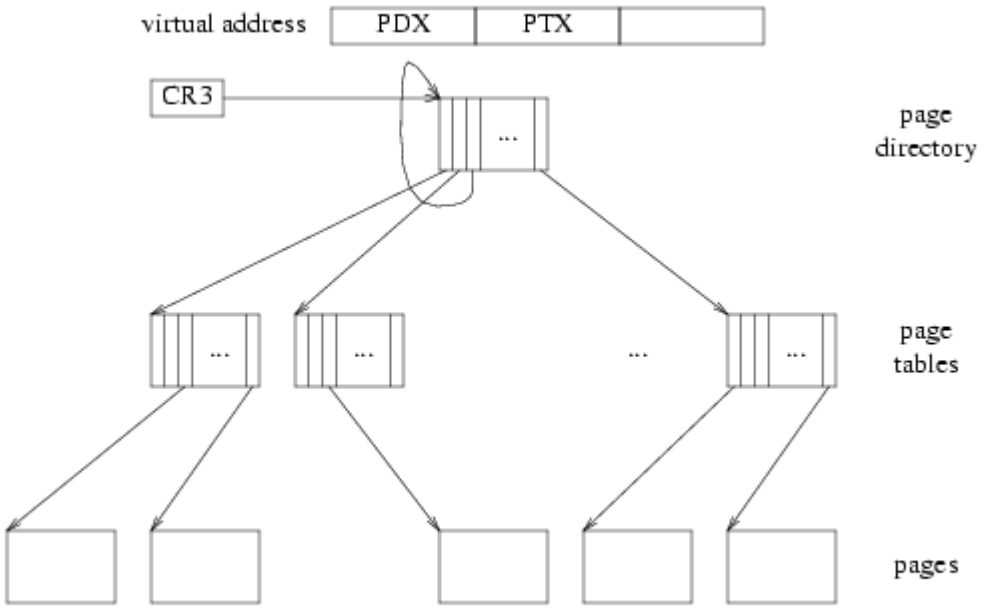


CR3 points at the page directory. The PDX part of the address indexes into the page directory to give you a page table. The PTX part indexes into the page table to give you a page, and then you add the low bits in.

But the processor has no concept of page directories, page tables, and pages being anything other than plain memory. So there's nothing that says a particular page in memory can't serve as two or three of these at once. The processor just follows pointers: pd = lcr3(); pt = *(pd+4*PDX); page = *(pt+4*PTX);

Diagramatically, it starts at CR3, follows three arrows, and then stops.

If we put a pointer into the page directory that points back to itself at index Z, as in



then when we try to translate a virtual address with PDX and PTX equal to V, following three arrows leaves us at the page directory. So that virtual page translates to the page holding the page directory. In Jos, V is 0x3BD, so the virtual address of the VPD is (0x3BD<<22)|(0x3BD<<12).

Now, if we try to translate a virtual address with PDX = V but an arbitrary PTX != V, then following three arrows from CR3 ends one level up from usual (instead of two as in the last case), which is to say in the page tables. So the set of virtual pages with PDX=V form a 4MB region whose page contents, as far as the processor is concerned, are the page tables themselves. In Jos, V is 0x3BD so the virtual address of the VPT is (0x3BD<<22).

So because of the "no-op" arrow we've cleverly inserted into the page directory, we've mapped the pages being used as the page directory and page table (which are normally virtually invisible) into the virtual address space.