Virtual Memory review, Initializing page tables for user processes

Virtual Memory Review

- VA = seg:offset. All instructions have default seg. The default seg can be overridden by specifying it explicitly.
- xv6 initializes all segs to 0:ffffffff, so we have a flat address space. All translation/protection done through paging. Segmentation is primarily used to specify the current privilege level (last two bits of CS register). Thus we need separate segment descriptors for user and kernel.
- The processor boots in physical address space (16-bit 8086 mode).
- As soon as it switches on 32-bit mode, segmentation hardware becomes active. However because the base of all segments is set to zero, the physical address is the same as the "offset".
- The kernel enables paging. As soon as the paging hardware becomes active, all addresses are translated through the page tables. In the first page table (`entrypgdir`), the kernel maps the first 4MB of VA space identically to the first 4MB of PA space. Assuming that the loaded kernel is less than 4MB, at the time of enabling paging, all virtual addresses are translated identically to physical addresses, and so everything works as before.
- In the first page table (`entrypgdir`), the kernel also maps the first 4MB starting at KERNBASE (0x80000000) to the first 4MB of PA space. This space above KERNBASE, will be the kernel's virtual address space in future.
- The kernel switches its EIP and ESP registers to point to the kernel's virtual address space. From then on, the kernel operates purely out of its virtual address space (above KERNBASE), and the identity mapping (below KERNBASE) can be removed and used for user processes.
- All symbols (variables, pointers) within the kernel image are given values in the kernel's virtual address space (above KERNBASE), so all future execution through de-referencing those variables will remain in the kernel address space.
- *Interesting Note*: Even the symbols in the entry code of the kernel (`entry.S`) were given virtual addresses (above 0x80000000) by the linker. However, when we jumped to the entry code from the bootloader, we ran it from its physical addresses. So this is the only exception, when code compiled for one address space (virtual address space) is run using its physical addresses.

Initializing page tables for user processes

```
big picture of xv6's virtual addressing scheme
  [diagram]
  0x00000000:0x80000000 -- user addresses below KERNBASE
  0x80000000:0x80100000 -- map low 1MB devices (for kernel)
  0x80100000:?          -- kernel instructions/data
  ?         :0x8E000000 -- 224 MB of DRAM mapped here
  0xFE000000:0x00000000 -- more memory-mapped devices

where does the paging h/w map these regions, in phys mem?
  [diagram]
  note double-mapping of user pages

main
  long sequence of calls to initialize subsystems
  first -- call kvmalloc to generate *another* page table

why another page table?
  map kernel only once, leaving space for low user memory
  map more physical memory, not just first 4 MB
  use 4KB pages instead of 4 MB pages

4 KB pages
  entrypgdir was simple: array of 1024 PDEs, each mapping 4 MB
  but 4 MB is too large a page size!
    very wasteful if you have small processes
    xv6 programs are a few dozen kilobytes
    4 MB pages require allocating full 4 MB of phys mem
  solution: x86 MMU supports 4 KB pages

how much space required for a flat page table w/ 4 KB pages?
  there are a million of them on a 32-bit machine
  32 bits per entry, so 4 MB for a full page table
  that also wastes lots of memory for small programs!
    you only need mappings for a few hundred pages
    so the rest of the million entries would be there but not needed

detailed structure of x86 page table
  see handout
  page table translates va to pa
    replacing top 20 bits in va with a [different] pa
    leaving bottom 12 bits alone
  logical view:
    a table of 32-bit PTEs
    2^20 PTEs, one for each 4096 bytes of address
    each PTE holds 20-bit pa of page
      or is marked invalid
    flags in low 12 bits
```

```
  implementation view:
    split page table into 4096-byte chunks ("page table pages")
    diagram
    each holds 1024 32-bit PTEs
    omit parts you don't need
    how to know where the parts of the table are? and which are valid?
  page-directory page
    1024 32-bit entries (PDE)
    each entry holds pa of a page-table page
      with entries for a range of 1024 pages
      or is blank (invalid)
    really top 20 bits of pa of page-table page
      pages are aligned, so bottom 12 bits always zero
    PDE holds some flags in those 12 bits
  %cr3 holds phys addr of page directory

how does x86 paging hardware translate a va?
  need to find the right PTE
  top 10 bits index PD to get address of PT
  next 10 bits index PT to get PTE
  flags in PTE
    P, W, U
    xv6 uses U to forbid user from using anything above 640 K



the page table is stored in DRAM
  thus xv6 has to allocate physical memory for PD and PTs
  each PD and PT fits in a physical page (4096 bytes) of memory

back to vm.c (sheet 17)
  main calls kvmalloc
  kvmalloc calls setupkvm

setupkvm
  creates a page table
  install mappings the kernel will need
  will be called once for each newly created process

setupkvm
  must allocate PD, allocate some PTs, put mappings in PTEs
  alloc allocates a physical page for the PD
    returns that page's virtual address above 0x8000000
    so we'll have to call V2P before installing in cr3
  memset so that default is no translation (no P bit)
  a call to mappages for each entry in kmap[]

what is kmap[]?
  an entry for each region of kernel virtual address space
  same as address space setup from earlier in lecture
  0x80000000 -> 0x00000000 (memory-mapped devices)
  0x80100000 -> 0x00100000 (kernel instructions and data)
  data -> data-0x80000000  (phys mem after where kernel was loaded)
  DEVSPACE -> DEVSPACE     (more memory mapped devices)
  note no mappings below va 0x80000000 -- future user memory there

mappages (sheet 16)
  (called by setupkvm for each kernel mapping range)
  arguments are PD, va, size, pa, perm
  adds mappings from a range of va's to corresponding pa's
  rounds b/c other uses pass in non-page-aligned addresses
  for each page-aligned address in the range
  call walkpgdir to find address of PTE
    need the PTE's address (not just content) b/c we want to modify

diagram of PD &c, as following steps build it

walkpgdir
  mimics how the paging h/w finds the PTE for an address
  refer to the handout
  a little complex b/c xv6 allocates page-table pages lazily
    might not be present, might have to allocate
  PDX extracts top ten bits
  &pgdir[PDX(va)] is the address of the relevant PDE
  now *pde is the PDE
  if PTE_P
    the relevant page-table page already exists
    PTE_ADDR extracts the PPN from the PDE
    p2v() adds 0x80000000, since PTE holds physical address
  if not PTE_P
    alloc a page-table page
```

```
    fill in PDE with PPN -- thus v2p
  now the PTE we want is in the page-table page
    at offset PTX(va)
    which is 2nd 10 bits of va

back to mappages
  put the desired pa into the PTE
  mark PTE as valid w/ PTE_P
  repeat for all pages in the range
```