```
6.828 2014 Lecture 6: Virtual Memory
==

* plan:
  address spaces
  paging hardware
  xv6 VM code

* today's problem:
  [user/kernel diagram]
  suppose the shell has a bug:
    sometimes it writes to a random memory address
  how can we keep it from wrecking the kernel?
    and from wrecking other processes?

* we want isolated address spaces
  each process has its own memory
  it can read and write its own memory
  it cannot read or write anything else

* xv6 uses x86's paging hardware to implement AS's
  ask questions! this material is important

* paging provides a level of indirection for addressing
  CPU -> MMU -> RAM
      VA     PA
  s/w can only ld/st to virtual addresses, not physical
  kernel tells MMU how to map each virtual address to a physical address
    MMU essentially has a table, indexed by va, yielding pa
    called a "page table"
  MMU can restrict what virtual addresses user code can use

* x86 maps 4-KB "pages"
  and aligned -- start on 4 KB boundaries
  thus page table index is top 20 bits of VA

* what is in a page table entry (PTE)?
  see handout
  top 20 bits are top 20 bits of physical address
    "physical page number"
    MMU replaces top 20 of VA with PPN
  low 12 bits are flags
    Present, Writeable, &c

* where is the page table stored?
  in RAM -- MMU loads (and stores) PTEs
  o/s can read/write PTEs

* would it be reasonable for page table to just be an array of PTEs?
  how big is it?
  2^20 is a million
  32 bits per entry
  4 MB for a full page table -- pretty big on early machines
  would waste 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

* x86 uses a "two-level page table" to save space
  diagram
  pages of PTEs in RAM
  page directory (PD) in RAM
  PDE also contains 20-bit PPN -- of a page of 1024 PTEs
  1024 PDEs point to PTE pages
    each PTE page has 1024 PTEs -- so 1024*1024 PTEs in total
```

```
    PD entries can be invalid
      those PTE pages need not exist
      so a page table for a small address space can be small

  * how does the mmu know where the page table is located in RAM?
    %cr3 holds phys address of PD
    PD holds phys address of PTE pages
    they can be anywhere in RAM -- need not be contiguous

  * how does x86 paging hardware translate a va?
    need to find the right PTE
    %cr3 points to PA of PD
    top 10 bits index PD to get PA of PT
    next 10 bits index PT to get PTE
    PPN from PTE + low-12 from VA

  * flags in PTE
    P, W, U
    xv6 uses U to forbid user from using kernel memory

  * what if P bit not set? or store and W bit not set?
    "page fault"
    CPU saves registers, forces transfer to kernel
    trap.c in xv6 source
    kernel can just produce error, kill process
    or kernel can install a PTE, resume the process
      e.g. after loading the page of memory from disk

  * Q: why mapping rather than e.g. base/bound?
    indirection allows paging h/w to solve many problems
    e.g. avoids fragmentation
    e.g. copy-on-write fork
    e.g. lazy allocation (home work for next lecture)
    many more techniques
    topic of next lecture

  * how does xv6 use the x86 paging hardware?

  * big picture of an xv6 address space -- one per process
    [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 xv6 map these regions, in phys mem?
    [diagram]
    note double-mapping of user pages

  * each process has its own address space
    and its own page table
    all processes have the same kernel (high memory) mappings
    kernel switches page tables (i.e. sets %cr3) when switching processes

  * Q: why this address space arrangement?
    user virtual addresses start at zero
      of course user va 0 maps to different pa for each process
    2GB for user heap to grow contiguously
      but needn't have contiguous phys mem -- no fragmentation problem
    both kernel and user mapped -- easy to switch for syscall, interrupt
    kernel mapped at same place for all processes
      eases switching between processes
    easy for kernel to r/w user memory
      using user addresses, e.g. sys call arguments
```

```
   easy for kernel to r/w physical memory
     pa x mapped at va x+0x80000000
     we'll see this soon while manipulating page tables

 * Q: what's the largest process this scheme can accomodate?

 * Q: could we increase that by increasing/decreasing 0x80000000?

 * To think about: does the kernel have to map all of phys mem
   into its virtual address space?

 * let's look at some xv6 virtual memory code
   virtual memory == address space / translation

 ** a process calls sbrk(n) to ask for n more bytes of heap memory
   malloc() uses sbrk()
   each process has a size
     kernel adds new memory at process's end, increases size
   sbrk() allocates physical memory (RAM)
   maps it into the process's page table
   returns the starting address of the new memory

 ** sys_sbrk() in sysproc.c
<!---
    trace sbrk from user space
    just run ls (or any other cmd from shell)
    the new process forked by shell calls malloc for execcmd structure
    malloc.c calls sbrk
-->


 ** growproc() in proc.c
   proc->sz is the process's current size
   allocuvm() does most of the work
   switchuvm sets %cr3 with new page table
     also flushes some MMU caches so it will see new PTEs

 ** allocuvm() in vm.c
   why if(newsz >= KERNBASE) ?
   why PGROUNDUP?
   arguments to mappages()...

 ** mappages() in vm.c
   arguments are PD, va, size, pa, perm
   adds mappings from a range of va's to corresponding pa's
   rounds b/c some 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
     put the desired pa into the PTE
     mark PTE as valid w/ PTE_P

 ** diagram of PD &c, as following steps build it

 ** walkpgdir() in vm.c
   mimics how the paging h/w finds the PTE for an address
   refer to the handout
   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
```