```
6.828 2016 Lecture 8: Page faults
==

* plan: cool things you can do with vm
  - Better performance/efficiency
    e.g., one zero-filled page
    e.g., copy-on-write fork
  - New features
    e.g., memory-mapped files

* virtual memory: several views
  * primary purpose: isolation
    each process has its own address space
  * Virtual memory provides a level-of-indirection
    provides kernel with opportunity to do cool stuff
        already some examples:
        - shared trampoline page
        - guard page
        but more possible...

* Key idea: change page tables on page fault
  Page fault is a form of a trap (like a system call)
  Xv6 panics on page fault
    But you don't have to panic!
  Instead:
    update page table instead of panic
        restart instruction (see userret() from traps lecture)
  Combination of page faults and updating page table is powerful!

* RISC-V page faults
  3 of 16 exceptions are related to paging
  Exceptions cause controlled transfers to kernel
    See traps lecture

* Information we might need at page fault to do something interesting:
  1) The virtual address that caused the fault
    See stval register; page faults set it to the fault address
  2) The type of violation that caused the fault
    See scause register value (instruction, load, and Store page fault)
  3) The instruction and mode where the fault occurred
    User IP: tf->epc
    U/K mode: implicit in usertrap/kerneltrap

* lazy/on-demand page allocation
  * sbrk() is old fashioned;
    applications often ask for memory they need
    - for example, the allocate for the largest possible input but
      an application will typically use less
    if they ask for much, sbrk() could be expensive
    - for example, if all memory is in use, have to wait until
      kernel has evicted some pages to free up memory
    sbrk allocates memory that may never be used.
  * moderns OSes allocate memory lazily
    plan:
      allocate physical memory when application needs it
      adjust p->sz on sbrk, but don't allocate
      when application uses that memory, it will result in page fault
      on pagefault allocate memory
      resume at the fault instruction
    may use less memory
      if not used, no fault, no allocation
    spreads the cost of allocation over the page faults instead
    of upfront in sbrk()
  * demo
```

```
    modify sysproc.c
    modify trap.c
    modify vm.c

* one zero-filled page (zero fill on demand)
  * applications often have large part of memory that must zero
    global arrays, etc.
    the "block starting symbol" (bbs) segment
  * thus, kernel must often fill a page with zeros
  * idea: memset *one* page with zeros
    map that page copy-on-write when kernel needs zero-filled page
    on write make copy of page and map it read/write in app address space

* copy-on-write fork
  * observation:
    xv6 fork copies all pages from parent (see fork())
    but fork is often immediately followed by exec
  * idea: share address space between parent and child
    modify fork() to map pages copy-on-write
      use extra available system bits (RSW) in PTEs
    on page fault, make copy of page and map it read/write

* demand paging
  * observation: exec loads the complete file into memory (see exec.c)
    expensive: takes time to do so (e.g., file is stored on a slow disk)
    unnecessary: maybe not the whole file will be used
  * idea: load pages from the file on demand
    allocate page table entries, but mark them on-demand
    on fault, read the page in from the file and update page table entry
    need to keep some meta information about where a page is located on disk
      this information is typically in structure called virtual memory area (VMA)
  * challenge: file larger than physical memory (see next idea)

* use virtual memory larger than physical memory
  * observation: application may need more memory than there is physical memory
  * idea: store less-frequently used parts of the address space on disk
    page-in and page-out pages of the address address space transparently
  * works when working sets fits in physical memory
    most popular replacement strategy: least-recently used (LRU)
    the A(cess) bit in the PTE helps the kernel implementing LRU
  * demo: run top and vmstat
    on laptop and dialup.athena.mit.edu
    see VIRT RES MEM SHR columns

* memory-mapped files
  * idea: allow access to files using load and store
    can easily read and writes part of a file
    e.g., don't have to change offset using lseek system call
  * Unix systems a new system call for m-mapped files:
    void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
  * kernel page-in pages of a file on demand
    when memory is full, page-out pages of a file that are not frequently used

* shared virtual memory
  * idea: allow processes on different machines to share virtual memory
    gives the illusion of physical shared memory, across a network
  * replicate pages that are only read
  * invalidate copies on write

* TLB management
  CPUs caches paging translation for speed
  xv6 flushes entire TLB during user/kernel transitions
    why?
  RISC-V allows more sophisticated plans
    * PTE_G: global TLB bits
```

```
      what page could use this?
    * ASID numbers
      TLB entries are tagged with ASID, so kernel can flush selectively
      SATP takes an ASID number
      sfence.vma also takes an ASID number
    * Large pages
      2MB and 1GB

* Virtual memory is still evolving
  Recent changes in Linux
    PKTI to handle meltdown side-channel
      (https://en.wikipedia.org/wiki/Kernel_page-table_isolation)
    xv6 basically implements KPTI
  Somewhat recent changes
    Support for 5-level page tables (57 address bits!)
    Support for ASIDs
  Less recent changes
    Support for large pages
    NX (No eXecute) PTE_X flag
```