

6.S081 2020 Lecture 4: Virtual Memory

==

- * plan:
 - address spaces
 - paging hardware
 - xv6 VM code

Virtual memory overview

- * today's problem:
 - [user/kernel diagram]
 - [memory view: diagram with user processes and kernel in memory]
 - 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
 - challenge:
 - how to multiplex several memories over one physical memory?
 - while maintaining isolation between memories
- * xv6 uses RISC-V's paging hardware to implement AS's
 - ask questions! this material is important
 - topic of next lab (and shows up in several other labs)
- * 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"
 - one page table per address space
 - MMU can restrict what virtual addresses user code can use
 - By programming the MMU, the kernel has complete control over va→pa mapping
 - Allows for many interesting OS features/tricks
- * RISC-V maps 4-KB "pages"
 - and aligned -- start on 4 KB boundaries
 - 4 KB = 12 bits
 - the RISC-V used in xv6 has 64-bit for addresses
 - thus page table index is top 64-12 = 52 bits of VA
 - except that the top 25 of the top 52 are unused
 - no RISC-V has that much memory now
 - can grow in future
 - so, index is 27 bits.
- * MMU translation
 - see Figure 3.1 of book
 - use index bits of VA to find a page table entry (PTE)
 - construct physical address using PPN from PTE + offset of VA
- * what is in PTE?
 - each PTE is 64 bits, but only 54 are used
 - top 44 bits of PTE are top bits of physical address
 - "physical page number"
 - low 10 bits of PTE flags
 - Present, Writeable, &c
 - note: size virtual addresses != size physical addresses
- * where is the page table stored?
 - in RAM -- MMU loads (and stores) PTEs
 - o/s can read/write PTEs
 - read/write memory location corresponding to PTEs
- * would it be reasonable for page table to just be an array of PTEs?
 - how big is it?

2²⁷ is roughly 134 million
64 bits per entry
134*8 MB for a full page table
wasting roughly 1GB per page table
one page table per address space
one address space per application
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

- * RISC-V 64 uses a "three-level page table" to save space
see figure 3.2 from book
page directory page (PD)
PD has 512 PTEs
PTEs point to another PD or is a leaf
so 512*512*512 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?
satp holds phys address of top PD
pages can be anywhere in RAM -- need not be contiguous
rewrite satp when switching to another address space/application
- * how does RISC-V paging hardware translate a va?
need to find the right PTE
satp register points to PA of top/L2 PD
top 9 bits index L2 PD to get PA of L1 PD
next 9 bits index L1 PD to get PA of L0 PD
next 9 bits index L0 PD to get PA of PTE
PPN from PTE + low-12 from VA
- * flags in PTE
V, R, W, X, U
xv6 uses all of them
- * what if V bit not set? or store and W bit not set?
"page fault"
forces transfer to kernel
trap.c in xv6 source
kernel can just produce error, kill process
in xv6: "usertrap(): unexpected scause ... pid=... sepc=... stval=..."
or kernel can install a PTE, resume the process
e.g. after loading the page of memory from disk
- * indirection allows paging h/w to solve many problems
e.g. phys memory doesn't have to be contiguous
avoids fragmentation
e.g. lazy allocation (a lab)
e.g. copy-on-write fork (another lab)
many more techniques
topic of next lecture
- * Q: why use virtual memory in kernel?
it is clearly good to have page tables for user processes
but why have a page table for the kernel?
could the kernel run with using only physical addresses?
top-level answer: yes
most standard kernels do use virtual addresses
why do standard kernels do so?
some reasons are lame, some are better, none are fundamental
 - the hardware makes it difficult to turn it off
e.g. on entering a system call, one would have to disable VM
 - the kernel itself can benefit from virtual addresses
mark text pages X, but data not (helps tracking down bugs)
unmap a page below kernel stack (helps tracking down bugs)
map a page both in user and kernel (helps user/kernel transition)

Virtual memory in xv6

- * kernel page table
See figure 3.3 of book

- simple mapping mostly
 - map virtual to physical one-on-one
- note double-mapping of trampoline
- note permissions
- why map devices?

- * each process has its own address space
 - and its own page table
 - see figure 3.4 of book
 - note: trampoline and trapframe aren't writable by user process
- kernel switches page tables (i.e. sets satp) 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
 - 16,777,216 GB for user heap to grow contiguously
 - but needn't have contiguous phys mem -- no fragmentation problem
 - both kernel and user map trampoline and trapframe page
 - eases transition user -> kernel and back
 - kernel doesn't map user applications
 - not easy for kernel to r/w user memory
 - need translate user virtual address to kernel virtual address
 - good for isolation (see spectre attacks)
 - easy for kernel to r/w physical memory
 - pa x mapped at va x

- * Q: does the kernel have to map all of phys mem into its virtual address space?

Code walk through

- * setup of kernel address space
 - kvmmap()
 - Q: what is address 0x10000000 (256M)
 - Q: how much address space does 1 L2 entry cover? (1G)
 - Q: how much address space does 1 L1 entry cover? (2MB)
 - Q: how much address space does 1 L0 entry cover? (4096)
 - print kernel page table
 - Q: what is size of address space? (512G)
 - Q: how much memory is used to represent it after 1st kvmmap()? (3 pages)
 - Q: how many entries is CLINT? (16 pages)
 - Q: how many entries is PLIC? (1024 pages, two level 1 PDs)
 - Q: how many pages is kernel text (8 pages)
 - Q: how many pages is kernel total (128M = 64 * 2MB)
 - Q: Is trampoline mapped twice? (yes, last entry and direct-mapped, entry [2, 3, 7])
 - kvmminithart();
 - Q: after executing w_satp() why will the next instruction be sfence_vma()?

- * mappages() in vm.c
 - arguments are top 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

- * walk() in vm.c
 - mimics how the paging h/w finds the PTE for an address
 - PX extracts the 9 bits at Level level
 - &pagetable[PX(level, va)] is the address of the relevant PTE
 - if PTE_V
 - the relevant page-table page already exists
 - PTE2PA extracts the PPN from the PDE
 - if not PTE_V
 - alloc a page-table page
 - fill in pte with PPN (using PA2PTE)
 - now the PTE we want is in the page-table page

- * procinit() in proc.c
 - alloc a page for each kernel stack with a guard page

- * setup user address space

allocproc(): allocates empty top-level page table
fork(): uvmcopy()
exec(): replace proc's page table with a new one
 uvmalloc
 loadseg
print user page table for sh
Q: what is entry 2?

- * a process calls sbrk(n) to ask for n more bytes of heap memory
user/umalloc.c calls sbrk() to get memory for the allocator
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
- * growproc() in proc.c
proc->sz is the process's current size
uvmalloc() does most of the work
when switching to user space satp will be loaded with updated page table
- * uvmalloc() in vm.c
why PGROUNDUP?
arguments to mappages()...