Our goal
Process and kernel memory

User memory (2GB)

Kernel memory (2GB)

Virtual

Process 1

Physical

Page table

Level 1

Level 2

Unused by xv6

0 - 4KB
4 - 8KB
...
2GB - 2GB + 4MB
...
(4MB-4K) - 4MB

0 - 4K

4K - 8K

Top of physical memory

0x80000000 (2GB, KERNBASE)
Multiple processes
Recap: our goal
Memory after boot
Locate page table directory entry
Allocate next level page table

pgtab = alloc()
Locate PTE entry
Locate PTE entry
Update mapping with physical addr

= p | mode
Move to next page
Walk page table

1754 walkpgdir(pde_t *pgdir, const void *va, int alloc)  
1755 {  
1756     pde_t *pde;  
1757     pte_t *pgtab;  
1758  
1759     pde = &pgdir[PDX(va)];  
1760     if(*pde & PTE_P){  
1761         pgtab = (pte_t*)P2V(PTE_ADDR(*pde));  
1762     } else {  
1763         if(!alloc || (pgtab = (pte_t*)kalloc()) == 0)  
1764             return 0;  
1765         // Make sure all those PTE_P bits are zero.  
1766         memset(pgtab, 0, PGSIZE);  
1767     }  
1768     *pde = V2P(pgtab) | PTE_P | PTE_W | PTE_U;  
1769 }  
1770 return &pgtab[PTX(va)];  
1771 }
mappages(pde_t *pgdir, void *va, uint size, uint pa, int perm)
{
    char *a, *last;
    pte_t *pte;

    a = (char*)PGROUNDDOWN((uint)va);
    last = (char*)PGROUNDDOWN(((uint)va) + size - 1);
    for(;;){
        if((pte = walkpgdir(pgdir, a, 1)) == 0)
            return -1;
        if(*pte & PTE_P)
            panic("remap");
        *pte = pa | perm | PTE_P;
        if(a == last)
            break;
        a += PGSIZE;
        pa += PGSIZE;
    }
    return 0;
}
Can a process overwrite kernel memory?
Privilege levels

- Each segment has a privilege level
  - DPL (descriptor privilege level)
  - 4 privilege levels ranging 0-3
Privilege levels

• Currently running code also has privilege level
  • “Current privilege level” (CPL): 0-3
  • Can access only less privileged segments
    – E.g., 0 can access 1, 2, 3

• Some instructions are “privileged”
  • Can only be invoked at CPL = 0
  • Examples:
    – Load GDT
    – MOV <control register>
      • E.g. reload a page table by changing CR3
Real world

- Only two privilege levels are used in modern OSes:
  - OS kernel runs at 0
  - User code runs at 3
- This is called “flat” segment model
  - Segments for both 0 and 3 cover entire address space
- But then... how the kernel is protected?
  - Page tables
Page table: user bit

- Each entry (both Level 1 and Level 2) has a bit
  - If set, code at privilege level 3 can access
  - If not, only levels 0-2 can access
- Note, only 2 levels, not 4 like with segments
- All kernel code is mapped with the user bit clear
  - This protects user-level code from accessing the kernel
TLB

- CPU caches results of page table walks
  - In translation lookaside buffer (TLB)
- Walking page table is slow
  - Each memory access is 200-300 cycles on modern hardware
  - L3 cache access is 70 cycles
TLB

- TLB is a cache (in CPU)
  - It is not coherent with memory
  - If page table entry is changes, TLB remains the same and is out of sync

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>0xf0231000</td>
<td>0x1000</td>
</tr>
<tr>
<td>0x00b31000</td>
<td>0x1f000</td>
</tr>
<tr>
<td>0xb0002000</td>
<td>0xc1000</td>
</tr>
</tbody>
</table>

Same Virt Addr. No Change!!!
Invalidating TLB

• After every page table update, OS needs to manually invalidate cached values

• Modern CPUs have “tagged TLBs”,
  • Each TLB entry has a “tag” – identifier of a process
  • No need to flush TLBs on context switch

• On Intel this mechanism is called
  • Process-Context Identifiers (PCIDs)
Questions?