6.828: Virtual Memory

Adam Belay
abelay@mit.edu
Outline

• Address spaces
• x86 Paging hardware
• xv6 VM code
• System call homework solutions
Today’s problem

Protection View:

SH

VI

CPL3

CPL0

Kernel
Today’s problem

Protection View:
- SH
- VI

Physical Memory View:
- $2^{32}$
- 0

CPL3

CPL0

Kernel
Goal: Isolation

• Each process has its own memory
• Can read and write its own memory
• But cannot read or write the kernel’s memory or another process’ memory
Solution: Introduce a level of indirection

- Plan: Software can only read and write to virtual memory
- Only kernel can program MMU
- MMU has a **page table** that maps virtual addresses to physical
- Some virtual addresses restricted to kernel-only
Virtual memory in x86

Virtual addresses are divided into 4-KB “pages”

Virtual Address:

- 31 - 12th bit: 20-bit page number
- 11 - 0th bit: 12-bit offset
## Page table entries (PTE)

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>AVL</th>
<th>G</th>
<th>P</th>
<th>A</th>
<th>T</th>
<th>D</th>
<th>A</th>
<th>C</th>
<th>W</th>
<th>U</th>
<th>W</th>
<th>P</th>
</tr>
</thead>
</table>

Some important bits:

- **Physical page number**: Identifies 20-bit physical page location; MMU replaces virtual bits with these physical bits
- **U**: If set, userspace (CPL3) can access this virtual address
- **W**: If set, the CPU can write to this virtual address
- **P**: If set, an entry for this virtual address exists
- **AVL**: Ignored by MMU
Strawman: Store PTEs in an array

GET_PTE(va) = &ptes[va >> 12]

<table>
<thead>
<tr>
<th>PPN</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How large is the array?
Strawman: Store PTEs in an array

GET_PTE(va) = &ptes[va >> 12]

<table>
<thead>
<tr>
<th>PPN</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

How large is the array?

- $2^{20} \times 32$ bits
- $2^{20} \times 4$ bytes
- 4 Megabytes!
x86 solution: Use two levels to save space

- 31: 10-bit DIR (1st level)
- 22-21: 10-bit TBL (2nd level)
- 12-11: 12-bit offset
x86 solution: Use two levels to save space

10-bit DIR (1st level)
Page Num FLG

10-bit TBL (2nd level)
Page Num FLG

Basically a tree!
What about a recursive mapping?
What about a recursive mapping?

- 10-bit DIR (1st level)
- 10-bit TBL (2nd level)
- 20-bit page table index

PPN

...
How do we program the MMU?

• %CR3 register is a pointer to current page table
• Hardware walks page table tree to find PTEs
• Recently used PTEs cached in TLB
Let’s talk more about flags

<table>
<thead>
<tr>
<th>Write Not Allowed</th>
<th>Read Not Allowed</th>
<th>Read Allowed</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Flags</td>
<td>PTE_P</td>
<td></td>
</tr>
<tr>
<td>Write Allowed</td>
<td>Not Possible</td>
<td>PTE_P</td>
</tr>
</tbody>
</table>

- If **PTE_U** is cleared, only the kernel can access
  - Why is this needed?
- What happens if flag permission is violated?
  - We get a page fault!
  - Then what happens?