Lecture 2, Problemset 2

Virtual Memory
Agenda

1. X86 Virtual Memory
2. xv6 Code Reading
3. Discussion of Problemset
X86 Virtual Memory Overview

- MMU supports:
  - Small pages (4 KBytes)
  - Large pages (4 MBytes)

- 2 level page table
  - Page directory (top level)
  - Page table (bottom level)
X86 Virtual Memory Translation

Diagram showing the process of virtual to physical address translation. The diagram includes components such as Page Directory (PDE), Page Table (PTE), Page Directory Number (VPN), and Page Offset (VPO). The process starts with the virtual address, which is broken down into different components, and then translated into a physical address.
X86 4KB Page Translation

Note: addresses in physical memory!
### X86 Page Directory Entry

<table>
<thead>
<tr>
<th>Empty</th>
<th>Ignored</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>4MB page</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bits 31:22 of address of 4MB page frame</td>
<td>0</td>
<td>Ign G 1 D A P C W T P U R / / SW 1</td>
</tr>
<tr>
<td>Page table</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bits 31:12 of address of page table</td>
<td>Ign</td>
<td>0 Ign A P C W T P U R / / SW 1</td>
</tr>
</tbody>
</table>

- Each PD has 1024 entries, 32 bits each
- 4 KBytes total size
- `#define PTE_* in mmu.h for xv6`
X86 Page Table Entry

- Each PT has 1024 entries, 32 bits each
- 4 KBytes total size
- `#define PTE_*` in `mmu.h` for `xv6`
X86 Translation Lookaside Buffer (TLB)

- CPU caches translation results after page walk
  - Cache is partially transparent, adding entries automatically
  - But does not track changes to page table

- Kernel needs to invalidate TLB manually
  - Only required when unmapping or changing permissions

- Two mechanisms for invalidation:
  - Flush: reloading cr3 (page directory base pointer)
  - invlpg instruction invalidates individual page

- Harder with multi-threaded processes on multiple cores
**X86 Page fault**

- Causes trap 14
- Also includes error code:

<table>
<thead>
<tr>
<th>Bit#</th>
<th>Label</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>P</td>
<td>Page table entry was valid</td>
</tr>
<tr>
<td>1</td>
<td>W</td>
<td>Caused by write access</td>
</tr>
<tr>
<td>2</td>
<td>U</td>
<td>Caused by user space access</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>Page table entry with reserved bits set</td>
</tr>
<tr>
<td>4</td>
<td>I</td>
<td>Caused by instruction fetch</td>
</tr>
</tbody>
</table>
xv6 Address Space Layout

Virtual

Device memory

Unused if less than 2 Gig of physical memory

Free memory

Kernel data

Kernel text

Program data & heap

Physical

Memory-mapped 32-bit I/O devices

Unused if less than 2 Gig of physical memory

At most 2 Gig

Extended memory

I/O space

Base memory

PHYSSTOP

4 Gig

0x100000

0x0

0x4000000

PAGESIZE

KERNBASE

+ 0x100000

end

0xFE000000

4 Gig
Code reading

- Kernel startup
- VM page table manipulation (vm.c)
- Page fault handling
- exec implementation
Problem set: Question 1

- Implement very simple `mmap` and `munmap`

```c
mmap(addr, length, rw, fd, offset)
munmap(addr, length)
```

- Map files into memory
  - Applications can read and write file using memory operations

- Implement the simplest case
  - Everything aligned, application picks address, file only mapped in one process

- Don't forget cleanup on exit
Problem set: Question 2

- Add demand paging for `mmap`
- Load pages "lazily" when accessed
  - Speeds up mapping large files
- You'll need to handle (some) page faults
  - And recover from them
- In problem set 4 you'll share files between processes