- Project 2a is due Friday
- Project 1b grades this week

- Discussion today: xv6 scheduler walk through for P2b
OFFICE HOURS

1. One question per student at a time
2. Please be prepared before asking questions
3. The TAs might not be able to fix your problem
4. Limited time per student

Search Piazza?
AGENDA / LEARNING OUTCOMES

Memory virtualization

How we reduce the size of page tables?
What can we do to handle large address spaces?
RECAP
For each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. check TLB for **VPN**
   - if miss:
     3. calculate addr of **PTE** (page table entry)
     4. read **PTE** from memory, add to TLB
3. extract **PFN** from TLB (page frame num)
4. build **PA** (phys addr)
5. read contents of **PA** from memory
TLB ACCESSES: EXAMPLE

1. extract **VPN**
2. check TLB for **VPN**
   if miss:
   3. calculate **PTE** addr
   4. read **PTE** from mem, add TLB
   5. extract **PPN** from TLB
   6. build **PA** (phys addr)
   7. read **PA** from memory

PT page table

<table>
<thead>
<tr>
<th>0 KB</th>
<th>4 KB</th>
<th>8 KB</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>PT</td>
<td>PT</td>
<td>P1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>P2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x0004</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x0000</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0x2000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

CPU's TLB

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

PTBR = 0x0000

PI pager table

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>9</td>
<td>...</td>
</tr>
</tbody>
</table>
TLB performance depends strongly on workload
  – Sequential workloads perform well
  – Workloads with temporal locality can perform well

TLBs increase cost of context switches
  – Flush TLB on every context switch
  – Add ASID to every TLB entry

In different systems, hardware or OS handles TLB misses
DISADVANTAGES OF PAGING

Additional memory reference to page table → Very inefficient
- Page table must be stored in memory
- MMU stores only base address of page table

Storage for page tables may be substantial
- Simple page table: Requires PTE for all pages in address space
  Entry needed even if page not allocated?
SMALLER PAGE TABLES
WHY ARE PAGE TABLES SO LARGE?

Waste!

Virt Mem

Phys Mem

code
heap

stack
Avoid Simple Linear Page Tables?

Use more complex page tables, instead of just big array
Any data structure is possible with software-managed TLB
  − Hardware looks for vpn in TLB on every memory access
  − If TLB does not contain vpn, TLB miss
    • Trap into OS and let OS find vpn → ppn translation
    • OS notifies TLB of vpn → ppn for future accesses

What about hardware managed TLBs?
OTHER APPROACHES

1. Segmented Pagetables
2. Multi-level Pagetables
   – Page the page tables
   – Page the pagetables of page tables…
3. Inverted Pagetables
**Valid PTEs are Contiguous**

<table>
<thead>
<tr>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>r-x</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>rw-</td>
</tr>
</tbody>
</table>

Note “hole” in addr space:
valids vs. invalids are clustered

How did OS avoid allocating holes in phys memory?

Segmentation
Combine Paging and Segmentation

Divide address space into segments (code, heap, stack)
  – Segments can be variable length
Divide each segment into fixed-sized pages.

Logical address divided into three portions

<table>
<thead>
<tr>
<th>seg # (4 bits)</th>
<th>page number (8 bits)</th>
<th>page offset (12 bits)</th>
</tr>
</thead>
</table>

Implementation

• Each segment has a page table
• Track base (physical address) and bounds of the page table per segment
## EXAMPLE: PAGING AND SEGMENTATION

<table>
<thead>
<tr>
<th>seg # (4 bits)</th>
<th>page number (8 bits)</th>
<th>page offset (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>seg</td>
<td>base</td>
<td>bounds</td>
</tr>
<tr>
<td>0</td>
<td>0x002000</td>
<td>0xff</td>
</tr>
<tr>
<td>1</td>
<td>0x000000</td>
<td>0x00</td>
</tr>
<tr>
<td>2</td>
<td>0x001000</td>
<td>0x0f</td>
</tr>
</tbody>
</table>

0x002070 read:
0x202016 read:
0x104c84 read:
0x010424 write:
0x210014 write:
0x203568 read:
ADVANTAGES OF PAGING AND SEGMENTATION

Advantages from using Segments
  – Decreases size of page tables. If segment not used, no need for page table

Advantages from using Pages
  – No external fragmentation
  – Segments can grow without any reshuffling

Advantages of using both
  – Increases flexibility of sharing
  – Share either single page or entire segment
Disadvantages of Paging and Segmentation

Potentially large page tables (for each segment)
  - Must allocate page table for each segment contiguously
  - More problematic with more address bits
  - Page table size?
    * Assume 2 bits for segment, 18 bits for page number, 12 bits for offset

Each page table is:
  = Number of entries * size of each entry
  = Number of pages * 4 bytes
  = $2^{18} \times 4$ bytes = $2^{20}$ bytes = 1 MB!
OTHER APPROACHES

1. Segmented Pagetables
2. Multi-level Pagetables
   - Page the page tables
   - Page the pagetables of page tables…
3. Inverted Pagetables
MULTILEVEL PAGE TABLES

Goal: Allow page table to be allocated non-contiguously

Idea: Page the page tables

- Creates multiple levels of page tables; outer level “page directory”
- Only allocate page tables for pages in use
- Used in x86 architectures (hardware can walk known structure)
20-bit address:

- outer page (4 bits)
- inner page (4 bits)
- page offset (12 bits)
How should logical address be structured? How many bits for each paging level?

Goal?

- Each inner page table fits within a page
- PTE size * number PTE = page size
  
  Assume PTE size = 4 bytes
  
  Page size = 2^{12} bytes = 4KB
  
  \[ \rightarrow \text{# bits for selecting inner page} = \]

Remaining bits for outer page:

- \[ 30 - \_ - \_ = \_ \text{ bits} \]
## Multilevel Translation Example

### Page Directory
```
<table>
<thead>
<tr>
<th>PPN</th>
<th>valid</th>
<th>PPN</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3</td>
<td>1</td>
<td>0x10</td>
<td>1</td>
</tr>
<tr>
<td>0x23</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x80</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x59</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0x92</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
```

### Page of PT (@PPN:0x3)
```
<table>
<thead>
<tr>
<th>PPN</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>1</td>
</tr>
</tbody>
</table>
```

### Page of PT (@PPN:0x92)
```
<table>
<thead>
<tr>
<th>PPN</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
```

**Translate 0x01ABC**

### 20-bit address:
- **Outer page**: 4 bits
- **Inner page**: 4 bits
- **Page offset**: 12 bits
Consider a virtual address space of 16KB with 64-byte pages.

1. How many bits will we have in our virtual address for this address space?

2. What is the total number of entries in the Linear Page Table for such an address space?

3. Consider a two-level page table now with a page directory. How many bits will be used to select the inner page assuming PTE size = 4 bytes?
### QUIZ12

#### page directory

<table>
<thead>
<tr>
<th>PPN</th>
<th>valid</th>
<th>PPN</th>
<th>valid</th>
<th>PPN</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x3</td>
<td>1</td>
<td>0x10</td>
<td>1</td>
<td>0x23</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x80</td>
<td>1</td>
<td>0x59</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0x92</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0x55</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x45</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

#### page of PT (@PPN:0x3)

<table>
<thead>
<tr>
<th>PPN</th>
<th>valid</th>
<th>PPN</th>
<th>valid</th>
<th>PPN</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10</td>
<td>1</td>
<td>0x23</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0x80</td>
<td>1</td>
<td>0x59</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>0x55</td>
<td>1</td>
</tr>
</tbody>
</table>

#### page of PT (@PPN:0x92)

<table>
<thead>
<tr>
<th>PPN</th>
<th>valid</th>
<th>PPN</th>
<th>valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x23</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0x59</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0x45</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

20-bit address:

- **outer page(4 bits)**
- **inner page(4 bits)**
- **page offset (12 bits)**

*translate 0xFEED0*
Problem with 2 levels?

Problem: page directories (outer level) may not fit in a page

Solution:

- Split page directories into pieces
- Use another page dir to refer to the page dir pieces.

How large is virtual address space with 4 KB pages, 4 byte PTEs,
(each page table fits in page)  

$4KB / 4 \text{ bytes} \rightarrow 1K \text{ entries per level}$

1 level:
2 levels:
3 levels:
FULL SYSTEM WITH TLBS

On TLB miss: lookups with more levels more expensive

Assume 3-level page table
Assume 256-byte pages
Assume 16-bit addresses
Assume ASID of current process is 211

<table>
<thead>
<tr>
<th>ASID</th>
<th>VPN</th>
<th>PFN</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>211</td>
<td>0xbb</td>
<td>0x91</td>
<td>1</td>
</tr>
<tr>
<td>211</td>
<td>0xff</td>
<td>0x23</td>
<td>1</td>
</tr>
<tr>
<td>122</td>
<td>0x05</td>
<td>0x91</td>
<td>1</td>
</tr>
<tr>
<td>211</td>
<td>0x05</td>
<td>0x12</td>
<td>0</td>
</tr>
</tbody>
</table>

How many physical accesses for each instruction? (Ignore ops changing TLB)

(a) 0xAA10: movl 0x1111, %edi

(b) 0xBB13: addl $0x3, %edi

(c) 0x0519: movl %edi, 0xFF10
INVERTED PAGE TABLE

Only store entries for virtual pages w/ valid physical mappings

Naïve approach:
   Search through data structure <ppn, vpn+asid> to find match
   Too much time to search entire table

Better:
   Find possible matches entries by hashing vpn+asid
   Smaller number of entries to search for exact match

Managing inverted page table requires software-controlled TLB
SUMMARY: BETTER PAGE TABLES

Problem: Simple linear page tables require too much contiguous memory

Many options for efficiently organizing page tables
If OS traps on TLB miss, OS can use any data structure
  – Inverted page tables (hashing)
If Hardware handles TLB miss, page tables must follow specific format
  – Multi-level page tables used in x86 architecture
  – Each page table fits within a page
SWAPPING
OS goal: Support processes when not enough physical memory
  – Single process with very large address space
  – Multiple processes with combined address spaces
User code should be independent of amount of physical memory
  – Correctness, if not performance

Virtual memory: OS provides illusion of more physical memory

Why does this work?
  – Relies on key properties of user processes (workload) and machine architecture (hardware)
Virtual Memory

Program
  code
  data
Leverage locality of reference within processes

- **Spatial:** reference memory addresses near previously referenced addresses
- **Temporal:** reference memory addresses that have referenced in the past
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

Implication:
- Process only uses small amount of address space at any moment
- Only small amount of address space must be resident in physical memory
Leverage memory hierarchy of machine architecture. Each layer acts as “backing store” for layer above.
SWAPPING INTUITION

Idea: OS keeps unreferenced pages on disk
   – Slower, cheaper backing store than memory

Process can run when not all pages are loaded into main memory
OS and hardware cooperate to make large disk seem like memory
   – Same behavior as if all of address space in main memory

Requirements:
   – OS must have **mechanism** to identify location of each page in address space → in memory or on disk
   – OS must have **policy** to determine which pages live in memory and which on disk
VIRTUAL ADDRESS SPACE MECHANISMS

Each page in virtual address space maps to one of three locations:
- Physical main memory: Small, fast, expensive
- Disk (backing store): Large, slow, cheap
- Nothing (error): Free

Extend page tables with an extra bit: present
- permissions (r/w), valid, present
- Page in memory: present bit set in PTE
- Page on disk: present bit cleared
  - PTE points to block on disk
  - Causes trap into OS when page is referenced
  - Trap: page fault
## What if access vpn 0xb?

### Phys Memory

### Disk

<table>
<thead>
<tr>
<th>PFN</th>
<th>valid</th>
<th>prot</th>
<th>present</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>r-x</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>rw-</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>rw-</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>rw-</td>
<td>1</td>
</tr>
</tbody>
</table>
VIRTUAL MEMORY MECHANISMS

First, hardware checks TLB for virtual address
   – if TLB hit, address translation is done; page in physical memory
Else ... 
   – Hardware or OS walk page tables
   – If PTE designates page is present, then page in physical memory (i.e., present bit is cleared)
Else
   – Trap into OS (not handled by hardware)
   – OS selects victim page in memory to replace
     • Write victim page out to disk if modified (use dirty bit in PTE)
   – OS reads referenced page from disk into memory
   – Page table is updated, present bit is set
   – Process continues execution
NEXT STEPS

Project 2a: Due Friday

Discussion section:
  Project 2b prep!
  xv6 scheduler walk through

Next class: More Swapping!