MEMORY: TLBS, SMALLER PAGETABLES

Shivaram Venkataraman
CS 537, Spring 2020
- Project 2a is due **Friday**
- Project 1b grades this week
- Midterm makeup emails
AGENDA / LEARNING OUTCOMES

Memory virtualization

What are the challenges with paging?
How we go about addressing them?
RECAP
Goal: Eliminate requirement that address space is contiguous

Idea:
Divide address spaces and physical memory into fixed-sized pages

Example page size: 4KB
PAGING TRANSLATION STEPS

For each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr)
2. calculate addr of **PTE** (page table entry)
3. read **PTE** from memory
4. extract **PFN** (page frame num)
5. build **PA** (phys addr)
6. read contents of **PA** from memory

14 bit addresses
Assume PT is at phys addr 0x5000
Assume PTE’s are 4 bytes
Assume 4KB pages – 12 bit offset

Simplified view of page table

| 2 | 0 | 80 | 99 |

READ 0x1100
**Pros/Cons of Paging**

**Pros**

No external fragmentation
- Any page can be placed in any frame in physical memory

Fast to allocate and free
- Alloc: No searching for suitable free space
- Free: Doesn’t have to coalesce with adjacent free space

**Cons**

Additional memory reference
- 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?
EXAMPLE: ARRAY ITERATOR

int sum = 0;
for (i=0; i<N; i++){
    sum += a[i];
}

Assume ‘a’ starts at 0x1000
Ignore instruction fetches
and access to ‘i’

What virtual addresses?
load 0x1000
load 0x1004
load 0x1008

What physical addresses?
load 0x0004
load 0x5000
load 0x0004
load 0x5004
load 0x0004
load 0x5008

What can you infer?
PTBR: 0x0000; PTE 4 bytes each
VPN 1 \(\rightarrow\) PPN 5
STRATEGY: CACHE PAGE TRANSLATIONS

- **CPU**
  - Translation Cache

- **RAM**
  - PT

---

memory interconnect
Fully associative
Any given translation can be anywhere in the TLB
Hardware will search the entire TLB in parallel
int sum = 0;
for (i = 0; i < 2048; i++) {
    sum += a[i];
}

Assume following virtual address stream:
load 0x1000
load 0x1004
load 0x1008
load 0x100C
...

Assume ‘a’ starts at 0x1000
Ignore instruction fetches and access to ‘i’

What will TLB behavior look like?
TLB ACCESSES: SEQUENTIAL EXAMPLE

CPU's TLB

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

PTBR

P1 pagetable

load 0x1000
load 0x1004
load 0x1008
load 0x100c
...
load 0x2000
load 0x2004
**TLB ACCESSES: SEQUENTIAL EXAMPLE**

**CPU's TLB**

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

**PTBR**

**P1 pagetable**

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Virt**

- load 0x1000
- load 0x1004
- load 0x1008
- load 0x100c
- load 0x2000
- load 0x2004
- load 0x2008
- load 0x200c
- load 0x5000
- load 0x5004
- load 0x5008
- load 0x500c
- load 0x0008
- load 0x4000
- load 0x4004
Consider a processor with 16-bit address space and 4kB page size. Assume Page Table is at 0x2000 and each PTE is of 4 bytes.

<table>
<thead>
<tr>
<th>VPN:0</th>
<th>0x0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0</td>
</tr>
<tr>
<td></td>
<td>0x1</td>
</tr>
<tr>
<td></td>
<td>0x9</td>
</tr>
<tr>
<td></td>
<td>0x7</td>
</tr>
<tr>
<td></td>
<td>0x8</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>VPN:15</td>
<td>0</td>
</tr>
</tbody>
</table>

Virtual Addresses
- 0x3000: load 0x5320, %eax
- 0x3004: load 0x4004, %ebx
- 0x3008: mul %ecx, %eax, %ebx
- 0x300C: store %ebx, 0x5324
- 0x3010: load 0x5328, %ebx

Memory accesses

Total
### Page Table

<table>
<thead>
<tr>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0</td>
<td>0x6</td>
</tr>
<tr>
<td>0x0</td>
<td>0x6</td>
</tr>
<tr>
<td>0x1</td>
<td>0x4</td>
</tr>
<tr>
<td>0x9</td>
<td></td>
</tr>
<tr>
<td>0x7</td>
<td>0x8</td>
</tr>
<tr>
<td>0x8</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### Virtual Addresses
- 0x3000: load 0x5320, %eax
- 0x3004: load 0x4004, %ebx
- 0x3008: mul %ecx, %eax, %ebx
- 0x300C: store %ebx, 0x5324
- 0x3010: load 0x5328, %ebx

### TLB

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>7</td>
<td>23</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>89</td>
</tr>
</tbody>
</table>
How to we replace entries in the TLB?

How do we handle context switches?
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}

Miss rate of TLB: \( \text{#TLB misses / #TLB lookups} \)

# TLB lookups? number of accesses to a =

# TLB misses? = number of unique pages accessed

Miss rate?

Would hit rate get better or worse with smaller pages?

Hit rate?
Workload A

```c
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

Workload B

```c
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
WORKLOAD ACCESS PATTERNS

Spatial Locality
Sequential Accesses

Temporal Locality
Repeated Random Accesses
TLB REPLACEMNT POLICIES

**LRU**: evict Least-Recently Used TLB slot when needed
Workload repeatedly accesses same offset (0x01) across 5 pages (strided access), but only 4 TLB entries.

What will TLB contents be over time?
How will TLB perform?
TLB REPLACEMENT POLICIES

LRU: evict Least-Recently Used TLB slot when needed

Random: Evict randomly chosen entry

Sometimes random is better than a “smart” policy!
CONTEXT SWITCHES

What happens if a process uses cached TLB entries from another process?

1. Flush TLB on each switch
   Costs: lose all recently cached translations

2. Track which entries are for which process
   - Address Space Identifier
   - Tag each TLB entry with an 8-bit ASID
TLB EXAMPLE WITH ASID

**TLB:**

<table>
<thead>
<tr>
<th>Valid</th>
<th>Virt</th>
<th>Phys</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>

**Virtual**

- load 0x1444 ASID: 12
- load 0x1444 ASID: 11

**Physical**

- P1 pagetable (ASID 11)
- P2 pagetable (ASID 12)
Context switches are expensive
Even with ASID, other processes “pollute” TLB

Architectures can have multiple TLBs
- 1 TLB for data, 1 TLB for instructions
- 1 TLB for regular pages, 1 TLB for “super pages”
HW AND OS ROLES

If H/W handles TLB Miss

CPU must know where pagetables are

- CR3 register on x86
- Pagetable structure fixed and agreed upon between HW and OS
- HW “walks” the pagetable and fills TLB

If OS handles TLB Miss:

“Software-managed TLB”

- CPU traps into OS upon TLB miss.
- OS interprets pagetables as it chooses
- Modify TLB entries with privileged instruction
TLB SUMMARY

Pages are great, but accessing page tables for every memory access is slow
Cache recent page translations → TLB
  – MMU performs TLB lookup on every memory access
TLB performance depends strongly on workload
  – Sequential workloads perform well
  – Workloads with temporal locality can perform well
In different systems, hardware or OS handles TLB misses
TLBs increase cost of context switches
  – Flush TLB on every context switch
  – Add ASID to every TLB entry
1. What problem(s) can be solved by using ASIDs?

2. For a hardware-managed TLB miss, which of the following statements are true?

3. For a software-managed TLB miss, which of the following statements are true?
DISADVANTAGES OF PAGING

Additional memory reference to page table \( \rightarrow \) 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
Many invalid PT entries

<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>28</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>rw-</td>
</tr>
</tbody>
</table>

How to avoid storing these?
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
OTHER APPROACHES

1. Segmented Pagetables (Today)
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>28</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

how to avoid storing these?
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
• Each segment track base (physical address) and bounds of the page table
### 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>

<table>
<thead>
<tr>
<th>0x002070</th>
<th>read:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x202016</td>
<td>read:</td>
</tr>
<tr>
<td>0x104c84</td>
<td>read:</td>
</tr>
<tr>
<td>0x010424</td>
<td>write:</td>
</tr>
<tr>
<td>0x210014</td>
<td>write:</td>
</tr>
<tr>
<td>0x203568</td>
<td>read:</td>
</tr>
</tbody>
</table>
ADVANTAGES OF PAGING AND SEGMENTATION

Advantages of Segments
– Supports sparse address spaces.
– Decreases size of page tables. If segment not used, not need for page table

Advantages of Pages
– No external fragmentation
– Segments can grow without any reshuffling
– Can run process when some pages are swapped to disk (next lecture)

Advantages of Both
– Increases flexibility of sharing
  • Share either single page or entire segment
  • How?
Disadvantages of Paging and Segmentation

Potentially large page tables (for each segment)
  - Must allocate each page table 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!!!
NEXT STEPS

Project 2a: Due Friday

Next class: Better pagetables, swapping!