MEMORY: TLBS, SMALLER PAGETABLES

Shivaram Venkataraman
CS 537, Spring 2019
- 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
## REVIEW: MATCH DESCRIPTION

<table>
<thead>
<tr>
<th>Description</th>
<th>Name of approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. one process uses RAM at a time</td>
<td></td>
</tr>
<tr>
<td>2. rewrite code and addresses before running</td>
<td></td>
</tr>
<tr>
<td>3. add per-process starting location to virt addr to obtain phys addr</td>
<td></td>
</tr>
<tr>
<td>4. dynamic approach that verifies address is in valid range</td>
<td></td>
</tr>
<tr>
<td>5. several base+bound pairs per process</td>
<td></td>
</tr>
</tbody>
</table>

Candidates: Segmentation, Static Relocation, Base, Base+Bounds, Time Sharing
FRAGMENTATION

Definition: Free memory that can’t be usefully allocated

Types of fragmentation
- External: Visible to allocator (e.g., OS)
- Internal: Visible to requester

Diagram:
- 0KB: Not Compacted
- 8KB: Operating System
- 16KB: (not in use)
- 24KB: Allocated
- 32KB: (not in use)
- 40KB: Allocated
- 48KB: (not in use)
- 56KB: Allocated
- 64KB: Internal
  - useful
  - free
Goal: Eliminate requirement that address space is contiguous
   Eliminate external fragmentation
   Grow segments as needed

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

Size: $2^n$, Example: 4KB
What is a good data structure?

Simple solution: Linear page table aka array
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 into register
MEMORY ACCESSES WITH Paging

0x0040: movl 0x1400, %edi

Assume PT is at phys addr 0x3000
Assume PTE's are 4 bytes
Assume 4KB pages
How many bits for offset? 12

Simplified view
of page table

| 2 | 0 | 3 | 1 |

Fetch instruction at logical addr 0x0040

• Access page table to get ppn for vpn __
• Mem ref 1:
  • Learn vpn __ is at ppn ___
  • Fetch instruction at _______ (Mem ref 2)

Exec, load from logical addr 0x1400

• Access page table to get ppn for vpn ___
• Mem ref 3:
  • Learn vpn ___ is at ppn ___
  • Movl from ______ into reg (Mem ref 4)
QUIZ: HOW BIG IS A PAGETABLE?

How big is a typical page table?
- assume 32-bit address space
- assume 4 KB pages
- assume 4 byte entries
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?
int sum = 0;
for (i=0; i<N; i++){
    sum += a[i];
}

Assume ‘a’ starts at 0x3000
Ignore instruction fetches
and access to ‘i’

What virtual addresses?
load 0x3000
load 0x3004
load 0x3008
load 0x300C

What physical addresses?
load 0x100C
load 0x7000
load 0x100C
load 0x7004
load 0x100C
load 0x7008
load 0x100C
load 0x700C
STRATEGY: CACHE PAGE TRANSLATIONS

- CPU
  - Translation Cache

- RAM
  - PT

memory interconnect
TLB: TRANSLATION LOOKASIDE BUFFER
TLB ORGANIZATION

TLB Entry

<table>
<thead>
<tr>
<th>Tag (virtual page number)</th>
<th>Physical page number (page table entry)</th>
</tr>
</thead>
</table>

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 'a' starts at 0x1000
Ignore instruction fetches and access to 'i'

What will TLB behavior look like?

Assume following virtual address stream:
load 0x1000
load 0x1004
load 0x1008
load 0x100C...
TLB ACCSESSES: SEQUENTIAL EXAMPLE

Virt

Phys

0x1000
0x1004
0x1008
0x100c
...
0x2000
0x2004

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>
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

PI pagetable

TLB Accesses: SEQUENTIAL Example

<table>
<thead>
<tr>
<th>Virt</th>
<th>Phys</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x1000</td>
<td>load 0x0004</td>
</tr>
<tr>
<td>load 0x1004</td>
<td>load 0x5000</td>
</tr>
<tr>
<td>load 0x1008</td>
<td>(TLB hit)</td>
</tr>
<tr>
<td>load 0x100c</td>
<td>load 0x5004</td>
</tr>
<tr>
<td>load 0x100c</td>
<td>(TLB hit)</td>
</tr>
<tr>
<td>load 0x2000</td>
<td>load 0x5008</td>
</tr>
<tr>
<td>load 0x2004</td>
<td>(TLB hit)</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>load 0x2004</td>
<td>load 0x500C</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>load 0x0008</td>
<td>load 0x0008</td>
</tr>
<tr>
<td>load 0x4000</td>
<td>(TLB hit)</td>
</tr>
<tr>
<td>load 0x4004</td>
<td>load 0x4004</td>
</tr>
</tbody>
</table>
PERFORMANCE OF TLB?

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

$\#\text{TLB lookups}?$ number of accesses to a =

$\#\text{TLB misses}? = \text{number of unique pages accessed}$

Miss rate?

Hit rate?

Would hit rate get better or worse with smaller pages?
TLB PERFORMANCE WITH WORKLOADS

Sequential array accesses almost always hit in TLB
  – Very fast!
What access pattern will be slow?
  – Highly random, with no repeat accesses
WORKLOAD ACCESS PATTERNS

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
WORKLOAD LOCALITY

Spatial Locality: future access will be to nearby addresses
Temporal Locality: future access will be repeats to the same data

What TLB characteristics are best for each type?
Spatial:
- Access same page repeatedly; need same vpn → ppn translation
- Same TLB entry re-used

Temporal:
- Access same address near in future
- Same TLB entry re-used in near future
- How near in future? How many TLB entries are there?
TLB REPLACEMENT POLICIES

**LRU**: evict Least-Recently Used TLB slot when needed

(More on LRU later in policies next week)

**Random**: Evict randomly choosen entry

Which is better?
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

(More on LRU later in policies next week)

Random: Evict randomly choosen entry

Sometimes random is better than a “smart” policy!
What happens if a process uses cached TLB entries from another process?

1. Flush TLB on each switch
   
   **Costly:** 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
     
     How many ASIDs do we get? Why not use PIDs?
**TLB Example with ASID**

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

**Physical**

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

**Page Tables**
- P1 pagetable (ASID 11)
- P2 pagetable (ASID 12)

**PTBR**: Physical to Page Table Base Register
TLB PERFORMANCE

Context switches are expensive
Even with ASID, other processes “pollute” TLB
  – Discard process A’s TLB entries for process B’s entries

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

Who Handles TLB MISS? **H/W** or **OS**?

**H/W**

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
HW AND OS ROLES

Who Handles TLB MISS? H/W or OS?

OS:

CPU traps into OS upon TLB miss
“Software-managed TLB”

OS interprets pagetables as it chooses
Modifying TLB entries is privileged
Need same protection bits in TLB as pagetable - rwx
Pages are great, but accessing page tables for every memory access is slow

Cache recent page translations → TLB
  – Hardware 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
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
QUIZ: HOW BIG ARE PAGE TABLES?

1. PTE’s are **2 bytes**, and **32** possible virtual page numbers

2. PTE’s are **2 bytes**, virtual addrs are **24 bits**, pages are **16 bytes**

3. PTE’s are **4 bytes**, virtual addrs are **32 bits**, and pages are **4 KB**

4. PTE’s are **4 bytes**, virtual addrs are **64 bits**, and pages are **4 KB**

How big is each page table?
WHY ARE PAGE TABLES SO LARGE?

Virt Mem

code
heap

stack

Phys Mem

Waste!
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>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>rw-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</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>

…many more invalid…

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
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></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>-</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?
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
# QUIZ: PAGING AND SEGMENTATION

<table>
<thead>
<tr>
<th>seg</th>
<th>base</th>
<th>bounds</th>
<th>R</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x002000</td>
<td>0xff</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0x000000</td>
<td>0x00</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0x001000</td>
<td>0x0f</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

- **0x002070 read:**
- **0x202016 read:**
- **0x104c84 read:**
- **0x010424 write:**
- **0x210014 write:**
- **0x203568 read:**
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:

\[ \text{Number of entries} \times \text{size of each entry} \]
\[ = \text{Number of pages} \times 4 \text{ bytes} \]
\[ = 2^{18} \times 4 \text{ bytes} = 2^{20} \text{ bytes} = 1 \text{ MB}!! \]
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 each page tables 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)
MULTILEVEL PAGE TABLES

30-bit address:

outer page (8 bits)  inner page (10 bits)  page offset (12 bits)

base of page directory
# Quiz: Multilevel

<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>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x23</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x80</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x59</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td></td>
<td></td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>0x92</td>
<td>0</td>
<td></td>
<td></td>
<td>0x55</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>0x45</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

20-bit address:

<table>
<thead>
<tr>
<th>outer page (4 bits)</th>
<th>inner page (4 bits)</th>
<th>page offset (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **translate 0x01ABC**
- **translate 0x00000**
- **translate 0xFEED0**
QUIZ: ADDRESS FORMAT FOR MULTILEVEL PAGING

30-bit address:

<table>
<thead>
<tr>
<th>outer page</th>
<th>inner page</th>
<th>page offset (12 bits)</th>
</tr>
</thead>
</table>

How should logical address be structured?

- How many bits for each paging level?

Goal?

- Each page table fits within a page
- PTE size * number PTE = page size
  - Assume PTE size = 4 bytes
  - Page size = $2^{12}$ bytes = 4KB
  - $2^2$ bytes * number PTE = $2^{12}$ bytes
  - $\rightarrow$ number PTE = $2^{10}$
- $\rightarrow$ # bits for selecting inner page = 10

Remaining bits for outer page:

- $30 - 10 - 12 = 8$ bits
Problem with 2 levels?

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

64-bit address:

outer page? | inner page (10 bits) | page offset (12 bits)

Solution:

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

VPN

PD idx 0 | PD idx 1 | PT idx | OFFSET

How large is virtual address space with 4 KB pages, 4 byte PTEs, each page table fits in page given 1, 2, 3 levels?

4KB / 4 bytes → 1K entries per level

1 level: 1K * 4K = $2^{22} = 4$ MB

2 levels: 1K * 1K * 4K = $2^{32} \approx 4$ GB

3 levels: 1K * 1K * 1K * 4K = $2^{42} \approx 4$ TB
QUIZ: 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

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

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

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

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

Only need 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
OTHER APPROACHES

1. Segmented Pagetables

2. Multi-level Pagetables
   - Page the page tables
   - Page the pagetables of page tables…

3. Inverted Pagetables
NEXT STEPS

Project 2a: Due Friday

Next class: Better pagetables, swapping!