MEMORY: SMALLER PAGETABLES

Shivaram Venkataraman
CS 537, Spring 2019
ADMINISTRIVIA

- Project 2a is due Friday
- Project 1b grades this week
- Midterm makeup
- Discussion today: xv6 scheduler walk through
OFFICE HOURS?
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. Up to 10 mins per student.

Search Piazza?
Discussion section: Using gdb
Extra office hours in the afternoon and evening tomorrow till 8pm!
Memory virtualization

How we reduce the size of page tables?
What can we do to handle large address spaces?
RECAP
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
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?
STRATEGY: CACHE PAGE TRANSLATIONS

Popular entries

memory interconnect

CPU
Translation Cache

RAM
PT
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
5. extract **PFN** from TLB (page frame num)
6. build **PA** (phys addr)
7. read contents of **PA** from memory
TLB ACCSESSES: SEQUENTIAL EXAMPLE

CPU's TLB

<table>
<thead>
<tr>
<th></th>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

PTB

PTE 4 bytes

0 KB

4 KB

8 KB

12 KB

16 KB

20 KB

24 KB

28 KB

PT

0

PT

P1

P2

P1

P2

P1 pagetable

0 1 2 3 ...

0x0000

0x0004

0x0008

0x000C

0x0010

0x0014

0x0018

0x001C

0x0200

VT

VPN

Phys

TLB miss

load 0x0000

load 0x3000

TLB hit

load 0x3004

load 0x0008

load 0x9000

Virt

offset
Pages are great, but accessing page tables for every memory access is slow. Cache recent page translations $\rightarrow$ 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 addr are 24 bits, pages are 16 bytes
   \[ 2^{20} \times 2 = 2^{21} \text{ bytes} \]

3. PTE’s are 4 bytes, virtual addr are 32 bits, and pages are 4 KB
   \[ 2^{20} \times 4 = 2^{22} = 4 \text{ MB} \]

4. PTE’s are 4 bytes, virtual addr are 64 bits, and pages are 4 KB
   \[ 2^{52} \times 4 = 2^{54} \]

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

Waste!

Virt Mem

Phys Mem

code

heap

stack

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>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

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
  • Each segment track base (physical address) and bounds of the page table
## Quiz: Paging and Segmentation

### Table: 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>

### Memory Addresses:

- `0x002070` read: `0x004 070`
- `0x202016` read: `0x003 016`
- `0x104c84` read: `0x004 070`
- `0x010424` write: `0x004 070`
- `0x210014` write: `0x004 070`
- `0x203568` read: `0x004 070`
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
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 \text{ bytes} = 2^{20} \text{ bytes} = 1 \text{ MB}!!!
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

Page Directory

4 KB

12 bits

10 bits
**MULTILEVEL**

- **page directory**
  - **PPN**: 0x3
  - **valid**: 1
  - Entries: 16

- **page of PT (@PPN: 0x3)**
  - **PPN**: 0x10
  - **valid**: 1
  - Entries:
    - 0x23
  - **PPN**: 0x80
  - **valid**: 0
  - Entries:
    - 0x59

- **page of PT (@PPN: 0x92)**
  - **PPN**:
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
    - 0
  - **valid**: 0

**20-bit address:**
- **outer page (4 bits)**
- **inner page (4 bits)**
- **page offset (12 bits)**

**translate 0x01ABC**
- **load 0x3**
- **PT offset 0**
- **outer page = 0**
- **inner page = 1**

**PPN = 0x23**
- **0x23 ABC**
- **4 KB**

**translate:**
- **offset**
**QUIZ: MULTILEVEL**

<table>
<thead>
<tr>
<th>page directory</th>
<th>page of PT (@PPN:0x3)</th>
<th>page of PT (@PPN:0x92)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN</td>
<td>valid</td>
<td>PPN</td>
</tr>
<tr>
<td>0x3</td>
<td>1</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>0x80</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>0x59</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>-</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>
</tbody>
</table>

20-bit address:

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

Translate 0xFEED0:

- Page directory (4 bits): 0xF
- Load PT 0x92
- Offset 0x55
- PPN = 0x55 0xED

Page directory:

- 0x92
- 0x10
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

  \[ \text{# bits for selecting inner page} = \frac{\text{Page size}}{\text{PTE size}} \]

  \[ \text{Remaining bits for outer page:} = 30 - 10 - 12 = 8 \text{ bits} \]
**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 bytes → 1K entries per level

1 level: $2^{22} = 2^{10} \cdot 2^{12} = 4 \text{ MB}$

2 levels: $4 \text{ GB}$

3 levels: $4 \text{ TB}$
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 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
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 been 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 for determining 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
<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>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>rw-</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>-</td>
<td>-</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>

What if access vpn 0xb?
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 (add dirty bit to 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:
   How to use gdb
   xv6 scheduler walk through

Next class: More Swapping!