CS 153
Design of Operating Systems
Spring 18

Lecture 17: Advanced Paging
Instructor: Chengyu Song
Slide contributions from
Nael Abu-Ghazaleh, Harsha Madhvasta and Zhiyun Qian

Some slides modified from originals by Dave O’hallaron
Recap: Paging Optimization

- Multi-level page tables (space)
- Caching translation results (TLBs) (time)
Intel Core i7 Memory System

**Processor package**

- **Core x4**
  - **Registers**
  - **Instruction fetch**
    - L1 d-cache: 32 KB, 8-way
    - L1 i-cache: 32 KB, 8-way
  - **L2 unified cache**
    - 256 KB, 8-way
  - **L1 d-TLB**
    - 64 entries, 4-way
  - **L1 i-TLB**
    - 128 entries, 4-way
  - **L2 unified TLB**
    - 512 entries, 4-way
  - **QuickPath interconnect**
    - 4 links @ 25.6 GB/s each
  - **DDR3 Memory controller**
    - 3 x 64 bit @ 10.66 GB/s
    - 32 GB/s total (shared by all cores)
  - **Main memory**
  - **MMU**
    - (addr translation)
  - **To other cores**
  - **To I/O bridge**
End-to-end Core i7 Address Translation

Virtual address (VA)

CPU

VPN

VPO

TLBT

TLBI

VPN1

VPN2

VPN3

VPN4

TLB miss

L1 TLB (16 sets, 4 entries/set)

VPN1

VPN2

VPN3

VPN4

TLB hit

L1 TLB (16 sets, 4 entries/set)

Page tables

CR3

PTE

PTE

PTE

PTE

L1 d-cache (64 sets, 8 lines/set)

L1 hit

Result

L2, L3, and main memory

L1 miss

Physical address (PA)

CT

CI

CO

CS153 – Lecture 16 – Paging
### Core i7 Level 1-3 Page Table Entries

<table>
<thead>
<tr>
<th>63</th>
<th>62</th>
<th>52</th>
<th>51</th>
<th>12</th>
<th>11</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>XD</td>
<td>Unused</td>
<td>Page table physical base address</td>
<td>Unused</td>
<td>G</td>
<td>PS</td>
<td>A</td>
<td>CD</td>
<td>WT</td>
<td>U/S</td>
<td>R/W</td>
<td>P=1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Available for OS (page table location on disk)**

**P=0**

---

Each entry references a 4K child page table

**P:** Child page table present in physical memory (1) or not (0).

**R/W:** Read-only or read-write access access permission for all reachable pages.

**U/S:** user or supervisor (kernel) mode access permission for all reachable pages.

**WT:** Write-through or write-back cache policy for the child page table.

**CD:** Caching disabled or enabled for the child page table.

**A:** Reference bit (set by MMU on reads and writes, cleared by software).

**PS:** Page size either 4 KB or 4 MB (defined for Level 1 PTEs only).

**G:** Global page (don’t evict from TLB on task switch)

**Page table physical base address:** 40 most significant bits of physical page table address (forces page tables to be 4KB aligned)
Today

- VM as a tool for memory management
  - Allocation
  - Sharing
  - Copy-on-Write
  - Memory mapped file
  - On-demand mapping

- VM as a tool for memory protection
VM as a Tool for Memory Management

- Key idea: each process has its own virtual address space
  - It can view memory as a simple linear array
  - Mapping function scatters addresses through physical memory
    » Well chosen mappings simplify memory allocation and management
Sharing

- Private virtual address spaces protect applications from each other
  - Usually exactly what we want

- But this makes it difficult to share data (have to copy)
  - Parents and children in a forking Web server or proxy will want to share an in-memory cache without copying

- We can use shared memory to allow processes to share data using direct memory references
  - Both processes see updates to the shared memory segment
    » Process B can immediately read an update by process A
Sharing (2)

- Sharing code and data among processes
  - Map virtual pages to the same physical page (here: PP 6)
Sharing (3)

- Can map shared memory at same or different virtual addresses in each process’ address space
  - Different:
    » 10th virtual page in P1 and 7th virtual page in P2 correspond to the 2nd physical page
    » Flexible (no address space conflicts), but pointers inside the shared memory segment are invalid
  - Same:
    » 2nd physical page corresponds to the 10th virtual page in both P1 and P2
    » Less flexible, but shared pointers are valid
Sharing (4)

- Linux API
  - Different address
    - `shm_open()`: create and open a new object, or open an existing object.
    - `mmap()`: map the shared memory object into the virtual address space of the calling process.
  - Same
    - `mmap()`: with MAP_SHARED
Copy on Write

- OSes spend a lot of time copying data
  - System call arguments between user/kernel space
  - Entire address spaces to implement fork()
- Use Copy on Write (CoW) to defer large copies as long as possible, hoping to avoid them altogether
  - Instead of copying pages, create shared mappings of parent pages in child virtual address space
  - Shared pages are protected as read-only in parent and child
    - Reads happen as usual
    - Writes generate a protection fault, trap to OS, copy page, change page mapping in client page table, restart write instruction
- How does this help fork()?
Execution of fork()

Parent process’s page table

Page 1
Page 2

Child process’s page table

Page 1
Page 2

Physical Memory
fork() with Copy on Write

When either process modifies Page 1, page fault handler allocates new page and updates PTE in child process.

Parent process’s page table

Protection bits set to prevent either process from writing to any page

Child process’s page table

Physical Memory
Simplifying Linking and Loading

- **Linking**
  - Each program has similar virtual address space
  - Code, stack, and shared libraries always start at the same address

- **Loading**
  - `execve()` allocates virtual pages for `.text` and `.data` sections
    - Creates PTEs marked as invalid
  - The `.text` and `.data` sections are copied, page by page, on demand by the virtual memory system

---

**Diagram**

- **Kernel virtual memory**
- **User stack** (created at runtime)
- **Memory-mapped region for shared libraries**
- **Run-time heap** (created by `malloc`)
- **Read/write segment** (.data, .bss)
- **Read-only segment** (.init, .text, .rodata)
- **Unused**

- **Memory invisible to user code**
- `%esp` (stack pointer)
- `brk` (boundary of readable memory)

- **Loaded from the executable file**
Mapped Files

- Mapped files enable processes to do file I/O using loads and stores
  - Instead of “open, read into buffer, operate on buffer, …”
- Bind a file to a virtual memory region (mmap() in Unix)
  - PTEs map virtual addresses to physical frames holding file data
  - Virtual address base + N refers to offset N in file
- Initially, all pages mapped to file are invalid
  - OS reads a page from file when invalid page is accessed
    - How?
Memory-Mapped Files

Pages are all invalid initially

Physical Memory

File Content 1

File Content 2

A read occurs

A read occurs

Page 1

Page 2

What happens if we unmap the memory?
How do we know whether we need to write changes back to file?
Writing Back to File

- OS writes a page to file when evicted, or region unmapped
- If page is not dirty (has not been written to), no write needed
  - Dirty bit trick (not protection bits)
Page Hit

- **Page hit**: reference to VM word that is in physical memory (DRAM cache hit)
Page Fault

- **Page fault**: reference to VM word that is not in physical memory (DRAM cache miss)
On-demand Mapping

- Allocate physical page
- Fix the page table
- Resume execution

How do we know whether the fault is fixable?
On-demand Mapping

- When the process calls mmap(), the kernel remembers
  - The region [addr, addr+length]
    » What virtual addresses are valid/mapped
  - The backing: just memory (ANONYMOUS) or file

- During page fault handling, the kernel checks
  - If the faulty virtual address is valid
  - If so, fix based on the backing
Today

- VM as a tool for memory management
  - Allocation
  - Sharing
  - Copy-on-Write
  - Memory mapped file
  - On-demand mapping

- VM as a tool for memory protection
VM as a Tool for Memory Protection

- Extend PTEs with permission bits
- Page fault handler checks these before remapping
  - If violated, send process SIGSEGV (segmentation fault)

<table>
<thead>
<tr>
<th>Process i:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 1:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
</tr>
<tr>
<td>VP 2:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Process j:</th>
<th>SUP</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>VP 0:</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
</tr>
<tr>
<td>VP 1:</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
</tr>
<tr>
<td>VP 2:</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
</tr>
</tbody>
</table>

Physical Address Space

- PP 2
- PP 4
- PP 6
- PP 8
- PP 9
- PP 11
Next time...

- Memory Hierarchy
  - No readings