Introduction to Computer Systems
15-213/18-243, spring 2009
18th Lecture, Mar. 24th

Instructors:
Gregory Kesden and Markus Püschel
Last Time: Address Translation

Virtual address

Virtual page number (VPN)  Virtual page offset (VPO)

Page table

Valid  Physical page number (PPN)

Valid bit = 0: page not in memory (page fault)

Physical address

Physical page number (PPN)  Physical page offset (PPO)

Page table base register (PTBR)

Page table address for process
Last Time: Page Fault

1. CPU (VA)
2. MMU
3. Cache/Memory
4. Exception
5. Page fault handler
6. Disk
7. CPU Chip

- 1. CPU accesses VA
- 2. MMU uses PTEA
- 3. Cache/Memory uses PTE
- 4. Exception触发
- 5. Page fault handler返回Victim page
- 6. Disk返回New page
A TLB hit eliminates a memory access
A TLB miss incurs an add’l memory access (the PTE)
Fortunately, TLB misses are rare
Today

- Virtual memory (VM)
  - Multi-level page tables
- Linux VM system
- Case study: VM system on P6
Multi-Level Page Tables

- **Given:**
  - 4KB \((2^{12})\) page size
  - 48-bit address space
  - 4-byte PTE

- **Problem:**
  - Would need a 256 GB page table!
    - \(2^{48} \times 2^{-12} \times 2^2 = 2^{38}\) bytes

- **Common solution**
  - Multi-level page tables
  - Example: 2-level page table
  - Level 1 table: each PTE points to a page table
  - Level 2 table: each PTE points to a page (paged in and out like other data)

- Level 1 table stays in memory
- Level 2 tables paged in and out
A Two-Level Page Table Hierarchy

Level 1 page table
- PTE 0
- PTE 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)
- PTE 8
- (1K - 9) null PTEs

Level 2 page tables
- PTE 0
- PTE 1
- PTE 2 (null)
- PTE 3 (null)
- PTE 4 (null)
- PTE 5 (null)
- PTE 6 (null)
- PTE 7 (null)

Virtual memory
- VP 0
- VP 1023
- VP 1024
- VP 1023
- Gap
- VP 2047
- VP 9215
- 1023 unallocated pages
- 6K unallocated VM pages
- 2K allocated VM pages for code and data
- 1 allocated VM page for the stack
- 1023 unallocated pages
- 6K unallocated VM pages
- 2K allocated VM pages for code and data
- 1 allocated VM page for the stack
Translating with a k-level Page Table

Virtual Address

Physical Address
Today

- Virtual memory (VM)
  - Multi-level page tables
- Linux VM system
- Case study: VM system on P6
Linux Organizes VM as Collection of “Areas”

- **pgd:**
  - Page directory address

- **vm_prot:**
  - Read/write permissions for this area

- **vm_flags**
  - Shared with other processes or private to this process
Linux Page Fault Handling

- **Is the VA legal?**
  - = Is it in an area defined by a `vm_area_struct`?
  - If not (#1), then signal segmentation violation

- **Is the operation legal?**
  - i.e., Can the process read/write this area?
  - If not (#2), then signal protection violation

- **Otherwise**
  - Valid address (#3): handle fault
Memory Mapping

- **Creation of new VM area** done via “memory mapping”
  - Create new vm_area_struct and page tables for area

- **Area can be backed by** (i.e., get its initial values from):
  - Regular file on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - Nothing (e.g., .bss)
    - First fault will allocate a physical page full of 0's (demand-zero)
    - Once the page is written to (dirtied), it is like any other page

- **Dirty pages are swapped back and forth between a special swap file.**

- **Key point:** no virtual pages are copied into physical memory until they are referenced!
  - Known as “demand paging”
  - Crucial for time and space efficiency
User-Level Memory Mapping

void *mmap(void *start, int len, int prot, int flags, int fd, int offset)

Disk file specified by file descriptor \texttt{fd}

Process virtual memory

len bytes

start (or address chosen by kernel)

len bytes

offset (bytes)
User-Level Memory Mapping

```c
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- **Map** len bytes starting at offset offset of the file specified by file description fd, preferably at address start
  - **start**: may be 0 for “pick an address”
  - **prot**: PROT_READ, PROT_WRITE, ...
  - **flags**: MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be start)

- **Example: fast file-copy**
  - Useful for applications like Web servers that need to quickly copy files.
  - `mmap()` allows file transfers without copying into user space.
mmap() Example: Fast File Copy

```c
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

/*
 * a program that uses mmap to copy
 * the file input.txt to stdout
 */

int main() {
    struct stat stat;
    int i, fd, size;
    char *bufp;
    
    /* open the file & get its size*/
    fd = open("./input.txt", O_RDONLY);
    fstat(fd, &stat);
    size = stat.st_size;
    
    /* map the file to a new VM area */
    bufp = mmap(0, size, PROT_READ,
                MAP_PRIVATE, fd, 0);
    
    /* write the VM area to stdout */
    write(1, bufp, size);
    exit(0);
}
```
Exec() Revisited

To run a new program $p$ in the current process using `exec()`:

- Free `vm_area_struct`'s and page tables for old areas

- Create new `vm_area_struct`'s and page tables for new areas
  - Stack, BSS, data, text, shared libraries.
  - Text and data backed by ELF executable object file
  - BSS and stack initialized to zero

- Set PC to entry point in `.text`
  - Linux will fault in code, data pages as needed
Fork() Revisited

- To create a new process using `fork()`:
  - Make copies of the old process’s `mm_struct`, `vm_area_struct`’s, and page tables.
    - At this point the two processes share all of their pages.
    - How to get separate spaces without copying all the virtual pages from one space to another?
      - “Copy on Write” (COW) technique.
  - Copy-on-write
    - Mark PTE's of writeable areas as read-only
    - Writes by either process to these pages will cause page faults
    - Flag `vm_area_struct`’s for these areas as private “copy-on-write”
      - Fault handler recognizes copy-on-write, makes a copy of the page, and restores write permissions.

- Net result:
  - Copies are deferred until absolutely necessary (i.e., when one of the processes tries to modify a shared page).
Memory System Summary

- **L1/L2 Memory Cache**
  - Purely a speed-up technique
  - Behavior invisible to application programmer and (mostly) OS
  - Implemented totally in hardware

- **Virtual Memory**
  - Supports many OS-related functions
    - Process creation, task switching, protection
  - Software
    - Allocates/shares physical memory among processes
    - Maintains high-level tables tracking memory type, source, sharing
    - Handles exceptions, fills in hardware-defined mapping tables
  - Hardware
    - Translates virtual addresses via mapping tables, enforcing permissions
    - Accelerates mapping via translation cache (TLB)
Further Reading

- Intel TLBs:
  - Application Note: “TLBs, Paging-Structure Caches, and Their Invalidation”, April 2007
Today

- Virtual memory (VM)
  - Multi-level page tables
- Linux VM system
- Case study: VM system on P6
Intel P6 (Bob Colwell’s Chip, CMU Alumni)

- **Internal designation for successor to Pentium**
  - Which had internal designation P5

- **Fundamentally different from Pentium**
  - Out-of-order, superscalar operation

- **Resulting processors**
  - Pentium Pro (1996)
  - Pentium II (1997)
    - L2 cache on same chip
  - Pentium III (1999)
    - The freshwater fish machines

- **Saltwater fish machines: Pentium 4**
  - Different operation, but similar memory system
  - Abandoned by Intel in 2005 for P6-based Core 2 Duo
P6 Memory System

32 bit address space
4 KB page size

L1, L2, and TLBs
- 4-way set associative

Inst TLB
- 32 entries
- 8 sets

Data TLB
- 64 entries
- 16 sets

L1 i-cache and d-cache
- 16 KB
- 32 B line size
- 128 sets

L2 cache
- unified
- 128 KB–2 MB
Review of Abbreviations

- **Components of the virtual address (VA)**
  - TLBI: TLB index
  - TLBT: TLB tag
  - VPO: virtual page offset
  - VPN: virtual page number

- **Components of the physical address (PA)**
  - PPO: physical page offset (same as VPO)
  - PPN: physical page number
  - CO: byte offset within cache line
  - CI: cache index
  - CT: cache tag
Overview of P6 Address Translation

- CPU
- VPN1, VPN2
- VPO
- TLBT, TLBI
- TLB (16 sets, 4 entries/set) - TLB miss
- L1 (128 sets, 4 lines/set) - L1 hit, L1 miss
- Page tables
- PDE, PTE
- PDBR
- PPN, PPO
- physical address (PA)
- L2 and DRAM

Virtual address (VA) --> TLB (16 sets, 4 entries/set) --> L1 (128 sets, 4 lines/set) --> physical address (PA)
P6 2-level Page Table Structure

- **Page directory**
  - 1024 4-byte page directory entries (PDEs) that point to page tables
  - One page directory per process.
  - Page directory must be in memory when its process is running
  - Always pointed to by PDBR

- **Page tables:**
  - 1024 4-byte page table entries (PTEs) that point to pages.
  - Page tables can be paged in and out.
# P6 Page Directory Entry (PDE)

<table>
<thead>
<tr>
<th>31</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></td>
<td>Avail</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>
</tr>
</tbody>
</table>

**Page table physical base address**: 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)

**Avail**: These bits available for system programmers

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

**PS**: page size 4K (0) or 4M (1)

**A**: accessed (set by MMU on reads and writes, cleared by software)

**CD**: cache disabled (1) or enabled (0)

**WT**: write-through or write-back cache policy for this page table

**U/S**: user or supervisor mode access

**R/W**: read-only or read-write access

**P**: page table is present in memory (1) or not (0)

<table>
<thead>
<tr>
<th>31</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Available for OS (page table location in secondary storage)</td>
<td>P=0</td>
</tr>
</tbody>
</table>
### P6 Page Table Entry (PTE)

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Page physical base address</td>
</tr>
<tr>
<td>12-11</td>
<td>Available, Global (G), Dirty (D), Accessed (A), Cache Disabled (CD), Write-through (WT)</td>
</tr>
<tr>
<td>9</td>
<td>User/Supervisor (U/S)</td>
</tr>
<tr>
<td>8</td>
<td>Read/Write (R/W)</td>
</tr>
<tr>
<td>7</td>
<td>Page is present in physical memory (P)</td>
</tr>
<tr>
<td>6-5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3-2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Page base address:** 20 most significant bits of physical page address (forces pages to be 4 KB aligned)

**Avail:** available for system programmers

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

**D:** dirty (set by MMU on writes)

**A:** accessed (set by MMU on reads and writes)

**CD:** cache disabled or enabled

**WT:** write-through or write-back cache policy for this page

**U/S:** user/supervisor

**R/W:** read/write

**P:** page is present in physical memory (1) or not (0)

**Available for OS (page location in secondary storage):** 

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Available, Page is present in physical memory (P)</td>
</tr>
</tbody>
</table>
Representations of VM Address Space

**Simplified Example**
- 16 page virtual address space

**Flags**
- P: Is entry in physical memory?
- M: Has this part of VA space been mapped?
P6 TLB Translation

CPU

VPN

TLBT

TLBI

virtual address (VA)

TLB (16 sets, 4 entries/set)

VPN1

VPN2

TLB hit

TLB miss

L1 hit

L1 miss

L1 (128 sets, 4 lines/set)

PDE

PTE

Page tables

physical address (PA)

PPN

PPO

32 result

L2 and DRAM

CT

CI

CO
P6 TLB

- TLB entry (not all documented, so this is speculative):
  - **V**: indicates a valid (1) or invalid (0) TLB entry
  - **TLBTag**: disambiguates entries cached in the same set
  - **PPN**: translation of the address indicated by index & tag
  - **G**: page is “global” according to PDE, PTE
  - **S**: page is “supervisor-only” according to PDE, PTE
  - **W**: page is writable according to PDE, PTE
  - **D**: PTE has already been marked “dirty” (once is enough)

- Structure of the data TLB:
  - 16 sets, 4 entries/set
1. Partition VPN into TLBT and TLBI.

2. Is the PTE for VPN cached in set TLBI?

3. Yes: Check permissions, build physical address

4. No: Read PTE (and PDE if not cached) from memory and build physical address
P6 TLB Translation

CPU

virtual address (VA)

VPN1

TLBT

virtual address (VA)

VPN2

TLBI

TLB (16 sets, 4 entries/set)

TLB miss

Page tables

PDE

PTE

L1 (128 sets, 4 lines/set)

L1 hit

L1 miss

physical address (PA)

PPN

PPO

CT

CI

CO

result

20

12

32

L2 and DRAM

CPU

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)
Translating with the P6 Page Tables (case 1/1)

- **Case 1/1:** page table and page present
- **MMU Action:**
  - MMU builds physical address and fetches data word
- **OS action**
  - None
Translating with the P6 Page Tables (case 1/0)

- **Case 1/0**: page table present, page missing

- **MMU Action:**
  - Page fault exception
  - Handler receives the following args:
    - %eip that caused fault
    - VA that caused fault
    - Fault caused by non-present page or page-level protection violation
      - Read/write
      - User/supervisor
Translating with the P6 Page Tables (case 1/0, cont.)

OS Action:
- Check for a legal virtual address.
- Read PTE through PDE.
- Find free physical page (swapping out current page if necessary)
- Read virtual page from disk into physical page
- Adjust PTE to point to physical page, set p=1
- Restart faulting instruction by returning from exception handler
Translating with the P6 Page Tables (case 0/1)

- Case 0/1: page table missing, page present
- Introduces consistency issue
  - Potentially every page-out requires update of disk page table
- Linux disallows this
  - If a page table is swapped out, then swap out its data pages too
Translating with the P6 Page Tables (case 0/0)

- Case 0/0: page table and page missing
- MMU Action:
  - Page fault
Translating with the P6 Page Tables (case 0/0, cont.)

OS action:
- Swap in page table
- Restart faulting instruction by returning from handler

Like case 0/1 from here on.
- Two disk reads
P6 L1 Cache Access

CPU

virtual address (VA)

VPN

VPO

TLBT

TLBI

TLB (16 sets, 4 entries/set)

VPN1

VPN2

TLB miss

TLB hit

L1 hit

L1 (128 sets, 4 lines/set)

PDE

PTE

Page tables

PPN

PPO

CT

CI

CO

physical address (PA)

result

L2 and DRAM

L1 miss

20

12

10

10

16

4

20

12

32

PDBR
L1 Cache Access

- Partition physical address: CO, CI, and CT
- Use CT to determine if line containing word at address PA is cached in set CI
- **No**: check L2
- **Yes**: extract word at byte offset CO and return to processor
**Observation**

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Generally we hit in TLB, so PPN bits (CT bits) available next
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible
x86-64 Paging

- **Origin**
  - AMD’s way of extending x86 to 64-bit instruction set
  - Intel has followed with “EM64T”

- **Requirements**
  - 48-bit virtual address
    - 256 terabytes (TB)
    - Not yet ready for full 64 bits
      - Nobody can buy that much DRAM yet
      - Mapping tables would be huge
      - Multi-level array map may not be the right data structure
  - 52-bit physical address
    - Requires 64-bit table entries
  - Keep traditional x86 4KB page size
    - \((4096 \text{ bytes per PT}) / (8 \text{ bytes per PTE}) = \text{ only 512 entries per page}\)
x86-64 Paging

Virtual address

VPN1 9  Page Map Table
VPN2 9  Page Directory Pointer Table
VPN3 9  Page Directory Table
VPN4 9  Page Table
VPO 12  

BR

PM4LE
PDPE
PDE
PTE

PPN 40
PPO 12

Physical address