Intel P6

Internal Designation for Successor to Pentium
- Which had internal designation P5

Fundamentally Different from Pentium
- Out-of-order, superscalar operation
- Designed to handle server applications
  - Requires high performance memory system

Resulting Processors
- PentiumPro (1996)
- Pentium II (1997)
  - Incorporated MMX instructions
    - Special instructions for parallel processing
  - L2 cache on same chip
- Pentium III (1999)
  - Incorporated Streaming SIMD Extensions
    - More instructions for parallel processing

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

Symbols:
- 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

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)

P6 Page Table Entry (PTE)
How P6 Page Tables Map Virtual Addresses to Physical Ones

Virtual address

<table>
<thead>
<tr>
<th>Virtual address</th>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>word offset into page directory</td>
<td>word offset into page table</td>
</tr>
<tr>
<td>VPN1</td>
<td>VPN2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VPN1</th>
<th>VPN2</th>
<th>VPO</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Physical address

<table>
<thead>
<tr>
<th>Physical address</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>

PDE | PDBR | PTE |

PDBR physical address of page directory

physical address of page base (if P=1)

PDE | PDBR | PTE |

PDBR physical address of page directory

physical address of page base (if P=1)

PT 0 | PT 1 | PT 2 | PT 3 |

Page Directory

<table>
<thead>
<tr>
<th>Page Directory</th>
</tr>
</thead>
<tbody>
<tr>
<td>P=1, M=1</td>
</tr>
<tr>
<td>P=1, M=1</td>
</tr>
</tbody>
</table>

Simplified Example

- 16 page virtual address space

Flags

- P: Is entry in physical memory?
- M: Has this part of VA space been mapped?

Representation of Virtual Address Space

TLB entry (not all documented, so this is speculative):

- V: indicates a valid (1) or invalid (0) TLB entry
- PD: is this entry a PDE (1) or a PTE (0)?
- tag: disambiguates entries cached in the same set
- PDE/PTE: page directory or page table entry

Structure of the data TLB:

- 16 sets, 4 entries/set

P6 TLB
Translating with the P6 TLB

1. Partition VPN into TLBT and TLBI.
2. Is the PTE for VPN cached in set TLBI?
   - Yes: then build physical address.
   - No: then read PTE (and PDE if not cached) from memory and build physical address.

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 but page missing.

MMU Action:
- page fault exception
- handler receives the following args:
  - 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 and copy to virtual page
- Restart faulting instruction by returning from exception handler.

Case 0/1: page table missing but 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 exception

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.
### P6 L1 Cache Access

![Diagram of P6 L1 Cache Access]

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

**Observation**
- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation taking place
- Then check with CT from physical address
- “Virtually indexed, physically tagged”
- Cache carefully sized to make this possible

---

### L1 Cache Access

**Partition physical address into CO, CI, and CT.**

**Use CT to determine if line containing word at address PA is cached in set CI.**

**If no: check L2.**

**If yes: extract word at byte offset CO and return to processor.**

---

### Speeding Up L1 Access

**Physical address (PA)**

- **CT**
- **CI**
- **CO**
- **PPN**
- **PPO**
- **Addr. Trans.**
- **No Change**

**virtual address (VA)**

- **VPN**
- **VPO**
- **20**
- **12**

**Tag Check**

---

### Linux Organizes VM as Collection of “Areas”

- **task_struct**
- **mm_struct**
- **vm_area_struct**
- **process virtual memory**

**pgd:**
- page directory address

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

**vm_flags**
- shared with other processes or private to this process

- **shared libraries**
- **data**
- **text**

0x040000000
0x0804a020
0x08048000

---

---
### Linux Page Fault Handling

**Is the VA legal?**
- i.e. is it in an area defined by a `vm_area_struct`?
- if not then signal segmentation violation (e.g., (1))

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

**If OK, handle fault**
- e.g., (3)

---

### 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)
  - nothing (e.g., bss)
- 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)
```

- `map len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start` (usually 0 for don’t care).
  - `prot`: MAP_READ, MAP_WRITE
  - `flags`: MAP_PRIVATE, MAP_SHARED
- return a pointer to the mapped area.
- 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.
Exec() Revisited

To run a new program $p$ in the current process using $\text{exec()}$:
- free $\text{vm\_area\_struct's}$ and page tables for old areas.
- create new $\text{vm\_area\_struct's}$ and page tables for new areas.
  - stack, bss, data, text, shared libc.
  - text and data backed by ELF executable object file.
  - bss and stack initialized to zero.
- set PC to entry point in .text
- Linux will swap in code and data pages as needed.

Fork() Revisited

To create a new process using $\text{fork()}$:
- make copies of the old process’s mm_struct, vm_area_struct’s, and page tables.
  - at this point the two processes are sharing all of their pages.
  - How to get separate spaces without copying all the virtual pages from one space to another?
    - “copy on write” technique.
- copy-on-write
  - make pages of writable areas read-only
  - flag vm_area_struct’s for these areas as private “copy-on-write”.
  - writes by either process to these pages will cause page faults.
    - 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

Cache Memory
- Purely a speed-up technique
- Behavior invisible to application programmer and OS
- Implemented totally in hardware

Virtual Memory
- Supports many OS-related functions
  - Process creation
    - Initial
    - Forking children
  - Task switching
  - Protection
- Combination of hardware & software implementation
  - Software management of tables, allocations
  - Hardware access of tables
  - Hardware caching of table entries (TLB)