Memory System
Case Studies
Oct. 31, 2007

Topics

- P6 address translation
- x86-64 extensions
- Linux memory management
- Linux page fault handling
- Memory mapping
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

- DRAM
- external system bus (e.g. PCI)
- L2 cache
- cache bus
- bus interface unit
- instruction fetch unit
- L1 i-cache
- inst TLB
- data TLB
- L1 d-cache
- L2 cache
- processor package

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

CPU

VPN  VPO
20  12

virtual address (VA)

TLBT  TLBI
16  4

TLB miss

VPN1  VPN2
10  10

TLB (16 sets, 4 entries/set)

PDE  PTE

Page tables

TLB (16 sets, 4 entries/set)

PDBR

PPN  PPO
20  12

physical address (PA)

L1 (128 sets, 4 lines/set)

L1 hit

L1 miss

result

L2 and DRAM

32

CT  CI  CO
20  7  5

Carnegie Mellon Qatar Campus
**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 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>Page table physical base addr</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</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>12 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>Available for OS (page table location in secondary storage)</td>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

P=0
## P6 Page Table Entry (PTE)

<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>Page physical base address</td>
<td>Avail</td>
<td>G</td>
<td>0</td>
<td>D</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>
</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)

<table>
<thead>
<tr>
<th>31</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available for OS (page location in secondary storage)</td>
<td>P=0</td>
</tr>
</tbody>
</table>
How P6 Page Tables Map Virtual Addresses to Physical Ones

- Virtual address
  - word offset into page directory
    - page directory
      - PDE
        - PDBR
          - physical address of page directory
      - VPN1
    - word offset into page table
      - page table
        - PTE
          - physical address of page base (if P=1)
    - 10
  - word offset into physical and virtual page
    - Physical address
      - PPN
        - 20
      - PPO
        - 12
Representation 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

virtual address (VA)

TLB (16 sets, 4 entries/set)

TLB miss

TLB hit

Page tables

CPU

VPN

VPO

16

4

TLBT

TLBI

VPN1

VPN2

10

10

12

20

PDE

PTE

PDBR

PPN

PPO

L1 hit

L1 miss

L1 (128 sets, 4 lines/set)

physical address (PA)

result

L2 and DRAM

32

CT

Ci

CO

20

7

5
P6 TLB

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

```
set 0
entry entry entry entry
entry entry entry entry
entry entry entry entry

set 1
entry entry entry entry
entry entry entry entry
entry entry entry entry

set 2
entry entry entry entry
entry entry entry entry
entry entry entry entry

set 15
entry entry entry entry
entry entry entry entry
entry entry entry entry```

---

P6 TLB

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

```
set 0
entry entry entry entry
entry entry entry entry
entry entry entry entry

set 1
entry entry entry entry
entry entry entry entry
entry entry entry entry

set 2
entry entry entry entry
entry entry entry entry
entry entry entry entry

set 15
entry entry entry entry
entry entry entry entry
entry entry entry entry```
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.
P6 Page Table Translation

CPU

virtual address (VA)

VPN
VPO

TLBT TLBI

TLB (16 sets, 4 entries/set)

TLB hit

VPN1 VPN2

TLB (16 sets, 4 entries/set)

TLB miss

Page tables

PDE PTE

PDBR

PTE

PPN PPO

CT CI CO

physical address (PA)

result

L1 hit

L1 (128 sets, 4 lines/set)

L1 miss

L2 and DRAM

virtual address (VA)
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.
Translating with the P6 Page Tables (case 0/1)

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

VPN1 VPN2

Mem

PDE|p=0

Page directory

Disk

PTE|p=0

Page table

data

Page page
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

virtual address (VA)

TLB (16 sets, 4 entries/set)

TLB hit

TLB miss

L1 (128 sets, 4 lines/set)

TLB (16 sets, 4 entries/set)

Page tables

physical address (PA)

CPU

VPN

VPO

TLBT TLBI

16 4

VPN1 VPN2

10 10

PDE PTE

PDBR

L1 hit

L1 miss

result

L2 and DRAM

20 12

32

PPN PPO

20 7 5

CT CI CO
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

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
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
- 52 bit physical address
  - Requires 64-bit table entries
- 4KB page size
  - Only 512 entries per page
x86-64 Paging

Virtual address

<table>
<thead>
<tr>
<th>9</th>
<th>9</th>
<th>9</th>
<th>9</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>VPN1</td>
<td>VPN2</td>
<td>VPN3</td>
<td>VPN4</td>
<td>VPO</td>
</tr>
</tbody>
</table>

- Page Map Table
- Page Directory Pointer Table
- Page Directory Table
- Page Table

BR

Physical address

<table>
<thead>
<tr>
<th>40</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>PPN</td>
<td>PPO</td>
</tr>
</tbody>
</table>
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

```
process virtual memory

shared libraries
0x40000000

data
0x0804a020

text
0x08048000

0

vm_area_struct

vm_end
vm_start
vm_prot
vm_flags
vm_next

vm_area_struct

vm_end
vm_start
vm_prot
vm_flags
vm_next

vm_area_struct

vm_end
vm_start
vm_prot
vm_flags
vm_next

mm_struct

mm

pgd

mmap

vm_next

task_struct

vm_next

vm_next
```

```
task_struct

mm

mm_struct

pgd

mmap

vm_next

vm_next

vm_next
```

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)
    » Initial page bytes come from a section of a file
  - Nothing (e.g., bss)
    » Initial page bytes are zeros
- 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
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.
mmap() Example: Fast File Copy

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

/* mmap.c - a program that uses mmap
 * to copy itself to stdout */

int main() {
    struct stat stat;
    int i, fd, size;
    char *bufp;

    /* open the file & get its size */
    fd = open("./mmap.c", 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);
}
```
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 libs.
  - 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 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 writeable 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 (mostly) OS
- Implemented totally in hardware

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