15-213
"The course that gives CMU its Zip!"

P6 / Linux Memory System
October 23, 2003

Topics
- P6 address translation
- Linux memory management
- Linux page fault handling
- Memory mapping
Intel P6
(Bob Collweil’s Chip, CMU Alumni)

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
IA32 Segmented VM Overview
Overview of P6 Address Translation

CPU → virtual address (VA) → TLB

TLB miss → PTE → PDBR

TLB hit → PTE → PDBR

virtual address (VA) → L1 (128 sets, 4 lines/set)

L1 hit → physical address (PA)

L1 miss → TLB (16 sets, 4 entries/set) → L1 (128 sets, 4 lines/set)

L1 hit → physical address (PA)

L2 and DRAM

20 12
VPN VPO

16 4
TLBT TLBI

10 10
VPN1 VPN2

PDE

Page tables

32 result
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 8 7 6 5 4 3 2 1 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page table physical base addr</td>
<td>Avail</td>
<td>G</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)

Available for OS (page table location in secondary storage) | P=0
# P6 Page Table Entry (PTE)

- **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>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>

<table>
<thead>
<tr>
<th>31</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Available for OS (page location in secondary storage)</td>
<td>P=0</td>
<td></td>
</tr>
</tbody>
</table>
How P6 Page Tables Map Virtual Addresses to Physical Ones

Virtual address

10
VPN1

word offset into page directory

page directory

PDE

PDBR

physical address of page directory

10
VPN2

word offset into page table

page table

PTE

physical address of page table base (if P=1)

12
VPO

word offset into physical and virtual page

physical address of page base (if P=1)

20
PPN

12
PPO

Physical address

– 10 –
# 4Mbyte PDE’s

## Page-Directory Entry (4-MByte Page)

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>30</td>
<td>29</td>
<td>28</td>
<td>27</td>
<td>26</td>
<td>25</td>
<td>24</td>
<td>23</td>
<td>22</td>
<td>21</td>
<td>20</td>
<td>19</td>
</tr>
<tr>
<td>Page Base Address</td>
<td>Reserved</td>
<td>PAT</td>
<td>Avail.</td>
<td>G</td>
<td>PS</td>
<td>D</td>
<td>A</td>
<td>P</td>
<td>C</td>
<td>D</td>
<td>W</td>
<td>T</td>
</tr>
</tbody>
</table>

- **Page Table Attribute Index**
- Available for system programmer’s use
- **Global page**
- **Page size (1 indicates 4 MBytes)**
- **Dirty**
- **Accessed**
- **Cache disabled**
- **Write-through**
- **User/Supervisor**
- **Read/Write**
- **Present**
Support for 4Mbyte Pages

Linear Address

31  22  21  0
Directory  Offset

Page Directory

10
Directory Entry

10
CR3 (PDBR)

*32 bits aligned onto a 4-KByte boundary.

4-MByte Page

Physical Address

1024 PDE = 1024 Pages
Representation of VM Address Space

**Page Directory**

- **PT 0**
  - P=0, M=1
  - P=0, M=0
  - P=0, M=1
  - P=0, M=0

- **PT 1**
  - P=0, M=1
  - P=0, M=0

- **PT 2**
  - P=0, M=1
  - P=0, M=0
  - P=0, M=1

- **PT 3**
  - P=1, M=1
  - P=1, M=0
  - P=1, M=1
  - P=0, M=1

**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

virtual address (VA)

VPN  VPO

TLBT  TLBI

TLB miss

TLB (16 sets, 4 entries/set)

VPN1  VPN2

PDE

PTE

Page tables

TLB hit

L1 hit

L2 and DRAM

L1 (128 sets, 4 lines/set)

L1 miss

PPN  PPO

physical address (PA)

15-213, F'03
P6 TLB

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

<table>
<thead>
<tr>
<th>32</th>
<th>16</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>PDE/PTE</td>
<td>Tag</td>
<td>PD</td>
<td>V</td>
</tr>
</tbody>
</table>

- **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

```
entry entry entry entry set 0
entry entry entry entry set 1
entry entry entry entry set 2
...
entry entry entry entry set 15
```
Translating with the P6 TLB

1. Partition VPN into TLBT and TLBI.

2. Is the PTE for VPN cached in set TLBI?
   - **3. Yes:** then build physical address.
   - **4. 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.
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

Diagram:
- Mem
  - VPN1, VPN2
    - PDE p=0
      - Page directory
        - PTE p=0
          - Page table
          - Data page
  - Disk
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

CPU

20 VPN 12 VPO

virtual address (VA)

16 4 TLBT TLBI

TLB miss

10 10 VPN1 VPN2

TLB (16 sets, 4 entries/set)

TLB hit

... PPN PPO

L1 (128 sets, 4 lines/set)

L1 hit

20 12

L1 miss

32 result

L2 and DRAM

PDBR

Page tables

physical address (PA)

15-213, F'03
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

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
Pentium 4 Xeon Changes

Pentium 4 Xeon / Pentium 4 Xeon MP

- 27 -
Linux Organizes VM as Collection of “Areas”

- **task_struct**
  - **mm_struct**
    - **mm**
    - **pgd**
    - **mmap**

- **vm_area_struct**
  - **vm_end**
  - **vm_start**
  - **vm_prot**
  - **vm_flags**
  - **vm_next**

- **process virtual memory**
  - **shared libraries**
    - Start Address: 0x40000000
  - **data**
    - Start Address: 0x0804a020
  - **text**
    - Start Address: 0x08048000

- **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?
- 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.
#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);
}
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 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 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)