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

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 byte 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
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>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
<td>Page Table Physical Base Address</td>
</tr>
<tr>
<td>12-11</td>
<td>Available, 0: P</td>
</tr>
<tr>
<td>9-8</td>
<td>Type (PTE)</td>
</tr>
<tr>
<td>7-0</td>
<td>Permissions, Access Control, Bit, Write, Global, Present</td>
</tr>
</tbody>
</table>

Page table physical base address: 20 most significant bits of physical page table address (since page tables are 4KB aligned)

Available: These bits available for system programmers

G: global page (don’t exist 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)

C: 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

RW: 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)
P6 Page Table Entry (PTE)

<table>
<thead>
<tr>
<th>31</th>
<th>12 11</th>
<th>9  8</th>
<th>7  6</th>
<th>5  4</th>
<th>3  2</th>
<th>1  0</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTE</td>
<td>Physical Base Address</td>
<td>Avail</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- Page base address: 30 most significant bits of physical page address (forces pages to be 4 KB aligned)
- Avail: available for system programmers
- G: global page (don’t exist from TLB on task switch)
- D: dirty (set by MMU on writes)
- A: accessed (set by MMU on reads and writes)
- CO: cache disabled or enabled
- WT: write-through or write-back cache policy for this page
- OS: user/supervisor
- RW: read/write
- P: page is present in physical memory (1) or not (0)

Available for OS (page location in secondary storage)

How P6 Page Tables Map Virtual Addresses to Physical Ones

Virtual address

- Word offset into page directory
- Word offset into page table
- Word offset into physical and virtual page

Physical address

- Physical address of page directory
- Physical address of page table (if P=1)

4Mbyte PDE’s

| 31 | 22 | 21 | 13 12 11 | 9  8 | 7  6 | 5  4 | 3  2 | 1  0 |
|----|----|----|--------|-----|-----|-----|-----|-----|-----|
| PTE | Page Directory Entry (4-MByte Page) | Page Base Address | Reserved | 4  | 3  | 2  | 1  | 0  |

- 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

<table>
<thead>
<tr>
<th>31</th>
<th>22</th>
<th>21</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directory</td>
<td>Offset</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- 4-MByte Page
- Directory Entry
- Physical Address

1024 PDE = 1024 Pages

- 32 bits aligned onto a 4-KByte boundary
**Representation of VM Address Space**

- **Page Directory**
- **PT 0**
- **PT 1**
- **PT 2**
- **Page 0**
- **Page 1**
- **Page 2**
- **Page 3**
- **Page 4**
- **Page 5**
- **Page 6**
- **Page 7**
- **Page 8**
- **Page 9**
- **Page 10**
- **Page 11**
- **Page 12**
- **Page 13**
- **Page 14**
- **Page 15**

**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**

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

- V: indicates a valid (1) or invalid (0) TLB entry
- P0: 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

---

**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 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

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

gpu

table

physical address (PA)

L1 hit

L1 (128 sets, 4 lines/set)

L2 and DRAM

L1 miss

L1 (16 sets, 4 entities/set)

TLB hit

TLB miss

virtual address (VA)

L1

L2

P6 CPU

Page tables

physical address (PA)
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

Linux Organizes VM as Collection of "Areas"
**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))

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., `bs`)
- if 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

**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` (usually 0 for don't care).
- flags: `MAP_READ`, `MAP_WRITE`
- 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
int main()
{
   struct stat stat;
   int i, fd, size;
   char *bufp;

   /* open the file and get its size*/
   fd = open("./map.c", O_RDONLY);
   fstat(fd, &stat);
   size = stat.st_size;
   /* map the file in a new VM area */
   bufp = mmap(NULL, size, PROT_READ,
               MAP_PRIVATE, fd, 0);
   /* write the VM area to stdout */
   write(fd, bufp, size);
   return 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, bs, data, text, shared libs.
  - text and data backed by ELF executable object file.
  - bs and stack initialized to zero.
  - set PC to entry point in .text
  - Linux will swap in code and data pages as needed.

**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)

**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).