15-213
“The course that gives CMU its Zip!”

P6/Linux Memory System
Oct. 31, 2002

Topics

- P6 address translation
- Linux memory management
- Linux page fault handling
- memory mapping
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

1. CPU
2. Virtual Address (VA)
3. TLB (16 sets, 4 entries/set)
4. Page tables
5. PDBR
6. PDE
7. PTE
8. L1 (128 sets, 4 lines/set)
9. L2 and DRAM
10. Physical Address (PA)

- TLB hit
- TLB miss
- L1 hit
- L1 miss
- L1 (128 sets, 4 lines/set)
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)**

<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 physical base address</td>
<td>Avail</td>
<td>G</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>1 0</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

Virtual page offset (VPO)

Virtual page number (VPN)

word offset into page directory

Page directory

Page directory entry (PDE)

physical address of page directory (if P=1)

physical address of page base (if P=1)

word offset into page table

Page table

Page table entry (PTE)

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

Physical address

physical address of page (if P=1)

word offset into physical and virtual page

Physical page offset (PPO)

Physical page number (PPN)
Representation of Virtual 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

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
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 miss

VPN1

VPN2

TLB hit

Page tables

PDE

PTE

PDBR

PPN

PPO

physical address (PA)

result

L2 and DRAM

L1 hit

L1 (128 sets, 4 lines/set)

L1 miss

CT

CI

CO
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

VPN 20
VPN1
VPN2

VPO

PDE p=0

PDBR

Page directory

Mem

Page table

PTE p=0

Disk

Data page

Data

disk

Mem

Page directory

Page table

PTE p=0

Disk

Data page

Data

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

VPN

VPO

TLBT TLBI

TLB (16 sets, 4 entries/set)

TLB (16 sets, 4 entries/set)

VPN1 VPN2

PDE

Page tables

PTE

PDBR

PTE

virtual address (VA)

TLB (16 sets, 4 entries/set)

PDE

PTE

virtual address (VA)

TLB (16 sets, 4 entries/set)

PDE

PTE

TLB miss

TLB (16 sets, 4 entries/set)

TLB hit

TLB hit

L1 (128 sets, 4 lines/set)

L1 hit

L1 miss

result

L2 and DRAM

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
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

Diagram:
- **task_struct**
- **mm_struct**
- **mm**
- **pgd**
- **mmap**

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

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

Address Ranges:
- 0x080400000
- 0x0804a020
- 0x08048000
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
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).
  - `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 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)