15-213

P6/Linux Memory System
October 31, 2000

Topics
• P6 address translation
• Linux memory management
• Linux page fault handling
• memory mapping

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 memory system

• 32 bit address space
• 4 KB pagesize
• 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 linesize
  • 128 sets
• L2 cache
  • unified
  • 128 KB -- 2 MB
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</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 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=1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Page table physical base address: 20 most significant bits of physical page table address (forces pages to be 4KB aligned)
Avail: 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
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</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)

Available for OS (page location in secondary storage) P=0

How P6 page tables map virtual addresses to physical ones

Virtual address

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

word offset into page directory

word offset into page table

Physical address

physical address of page base (if P=1)

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

P6 TLB translation

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

P6 page table translation

P6 page table translation

TLB (16 sets, 4 entries/set)
**Translating with the P6 page tables (case 1/1)**

**Case 1/1: page table and page present.**

**MMU Action:**
- MMU build physical address and fetch 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.

**Does Linux disallow 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

OS action:
• swap in page table.
• restart faulting instruction by returning from handler.

Like case 0/1 from here on.

Translating with the P6 page tables
(case 0/0, cont)

P6 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.
Linux organizes VM as a 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

Memory mapping

Creating a new VM area is 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

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 and 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_structs` and page tables for old areas.
- create new `vm_area_structs` 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_structs`, 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_structs` 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).