

Intel P6 Virtual Memory

CSE 361S

Intel P6

Internal Designation for Successor to Pentium

- Which had internal designation P5

Fundamentally Different from Pentium

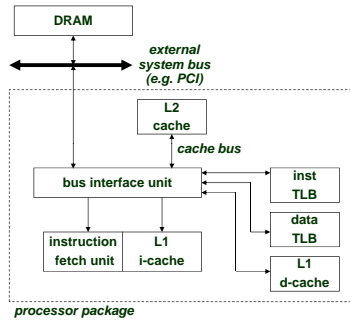
- Out-of-order, superscalar operation
- Designed to handle server applications
 - Requires high performance memory system

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

-2-

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

-3-

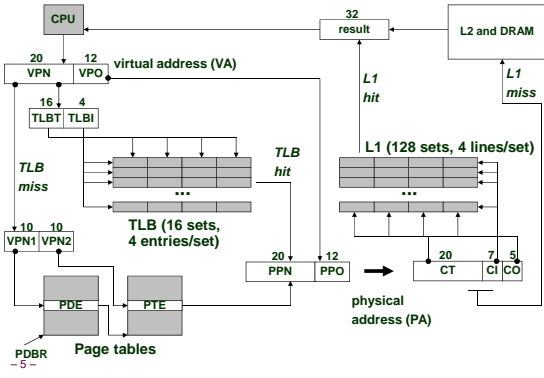
Review of Abbreviations

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

-4-

Overview of P6 Address Translation

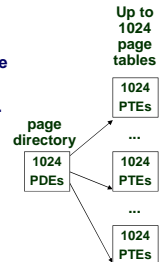


-5-

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.

-6-

P6 Page Directory Entry (PDE)

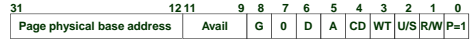


Page table physical base address: 20 most significant bits of physical page table address (forces page tables to be 4KB aligned)
Avail: These bits available for system programmers
G: global page (don't evict from TLB on task switch)
PS: page size 4K (0) or 4M (1)
A: accessed (set by MMU on reads and 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)



- 7 -

P6 Page Table Entry (PTE)

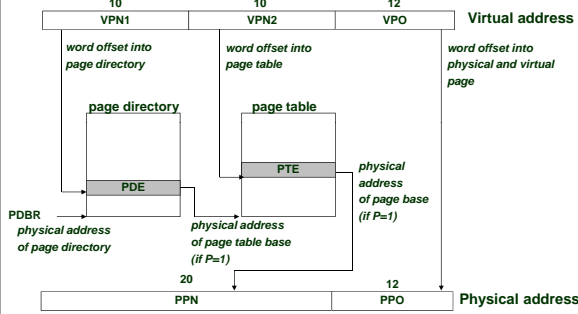


Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)
Avail: available for system programmers
G: global page (don't evict from TLB on task switch)
D: dirty (set by MMU on writes)
A: accessed (set by MMU on reads and writes)
CD: cache disabled or enabled
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)



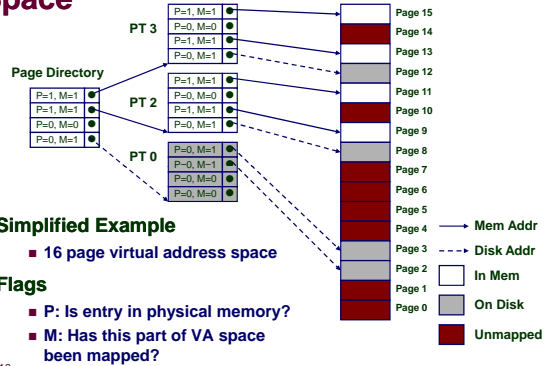
- 8 -

How P6 Page Tables Map Virtual Addresses to Physical Ones



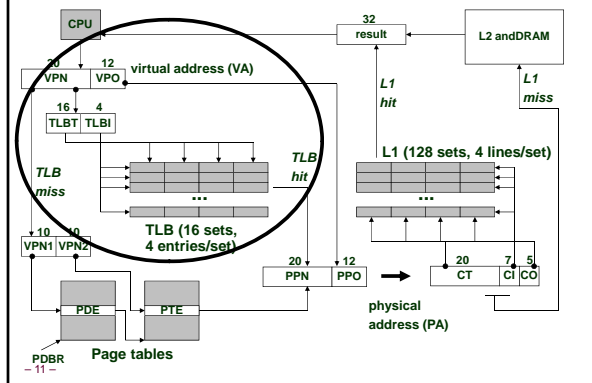
- 9 -

Representation of Virtual Address Space



- 10 -

P6 TLB Translation



- 11 -

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



- 12 -

Translating with the P6 TLB

- Partition VPN into TLBT and TLBI.
- 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.

- 13 -

P6 page table translation

- 14 -

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

- 15 -

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

- 16 -

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.

- 17 -

Translating with the P6 Page Tables (case 0/1)

Case 0/1: page table missing but page present.

Introduces consistency issue.

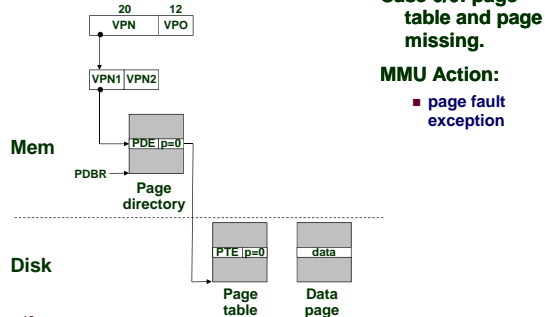
- potentially every page out requires update of disk page table.

Linux disallows this

- if a page table is swapped out, then swap out its data pages too.

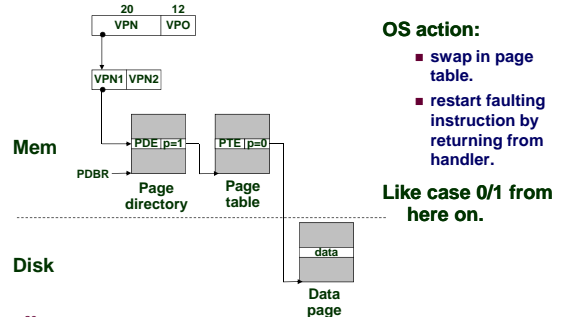
- 18 -

Translating with the P6 Page Tables (case 0/0)



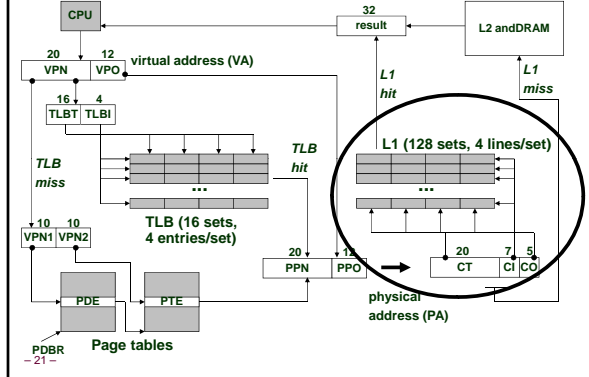
- 19 -

Translating with the P6 Page Tables (case 0/0, cont)



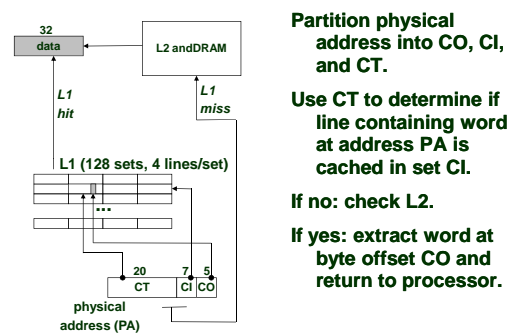
- 20 -

P6 L1 Cache Access



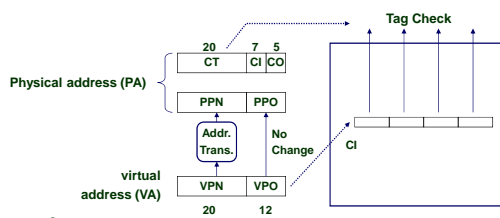
- 21 -

L1 Cache Access



- 22 -

Speeding Up L1 Access

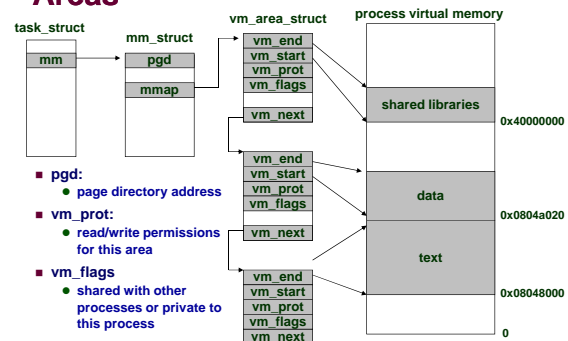


Observation

- 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

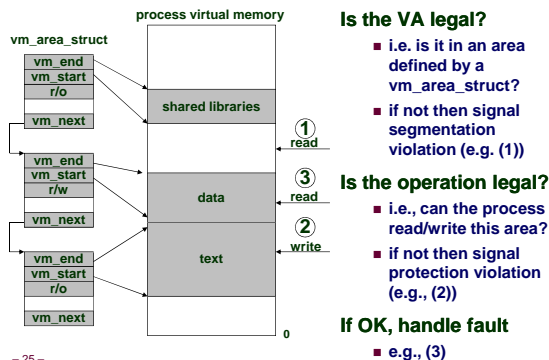
- 23 -

Linux Organizes VM as Collection of "Areas"



- 24 -

Linux Page Fault Handling



- 25 -

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

- 26 -

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- map `len` bytes starting at `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.

- 27 -

mmap() Example: Fast File Copy

```
#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);
}
```

- 28 -

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)

- 29 -