15-213

"The course that gives CMU its Zip!"

Pentium/Linux Memory System March 17, 2005

Topics

- P6 address translation
- x86-64 extensions
- Linux memory management
- Linux page fault handling
- Memory mapping

`17-linuxmem.ppt`

Intel P6

(Bob Colwell's Chip, CMU Alumni)

Internal Designation for Successor to Pentium

- Which had internal designation P5

Fundamentally Different from Pentium

- Out-of-order, superscalar operation

Resulting Processors

- PentiumPro (1996)
- Pentium II (1997)
  - L2 cache on same chip
- Pentium III (1999)
  - The Fish machines

Current Systems Pentium 4

- Different operation, but similar memory system

Legacy Issue

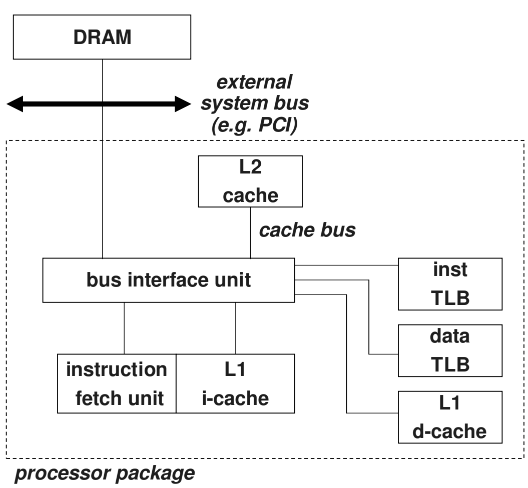
x86 Processors Support Multiple Addressing Schemes

- Segment-based
  - Variable sized blocks
  - Numerous variants
    - » Real mode
    - » Virtual-8086 mode
    - » Protected mode
  - All obsolete
- Page-based
  - Fixed size blocks
  - Virtual address: 32b
  - Physical address: 32b, 40b
  - Page sizes: 4K, 2M, 4M

What Will We Do?

- Look at the common case

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

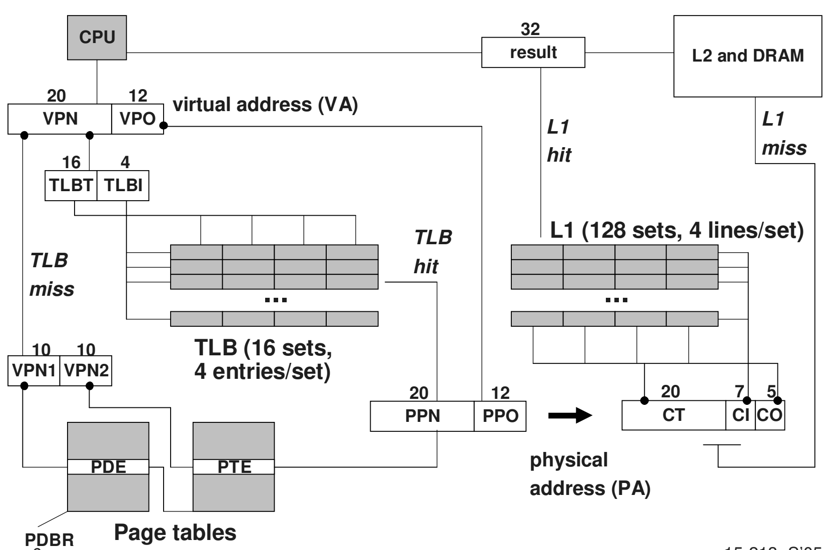
L2 cache
- unified
- 128 KB -- 2 MB

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



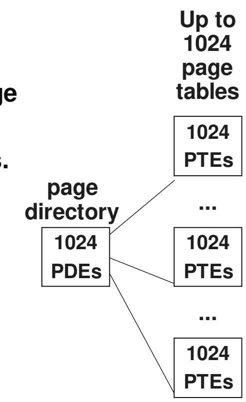
P6 2-level Page Table Structure

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)

31–12	11–9	8	7	6	5	4	3	2	1	0
Page table physical base addr	Avail	G	PS		A	CD	WT	U/S	R/W	P=1

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)

31–1	0
Available for OS (page table location in secondary storage)	P=0

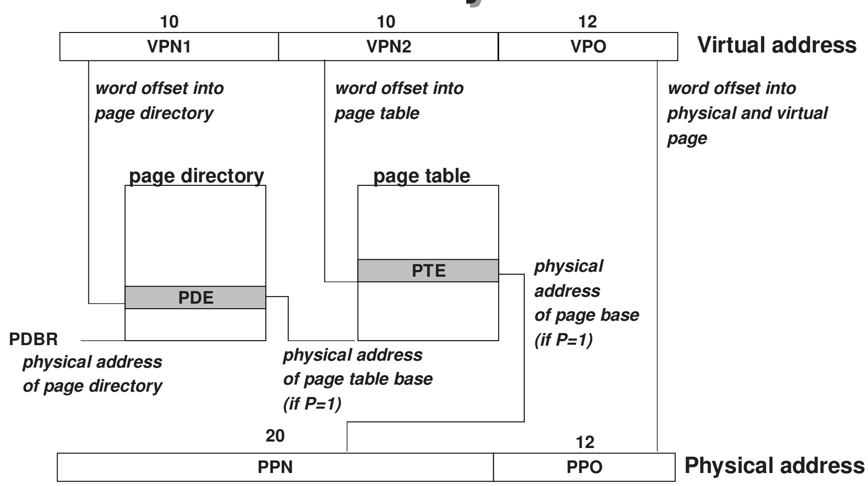
P6 Page Table Entry (PTE)

31 – 12	11 – 9	8	7	6	5	4	3	2	1	0
Page physical base address	Avail	G	0	D	A	CD	WT	U/S	R/W	P=1

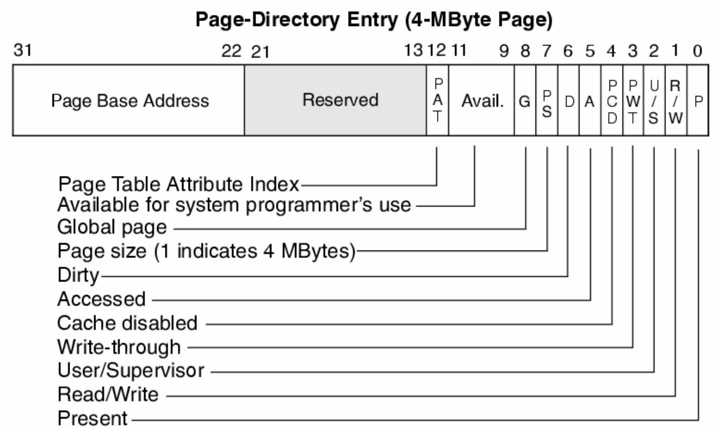
- Page base address: 20 most significant bits of physical page address (forces pages to be 4 KB aligned)
- Avail: available for system programmers
- G: global page (don't evict from TLB on task switch)
- D: dirty (set by MMU on writes)
- A: accessed (set by MMU on reads and writes)
- CD: cache disabled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

31 – 1	0
Available for OS (page location in secondary storage)	P=0

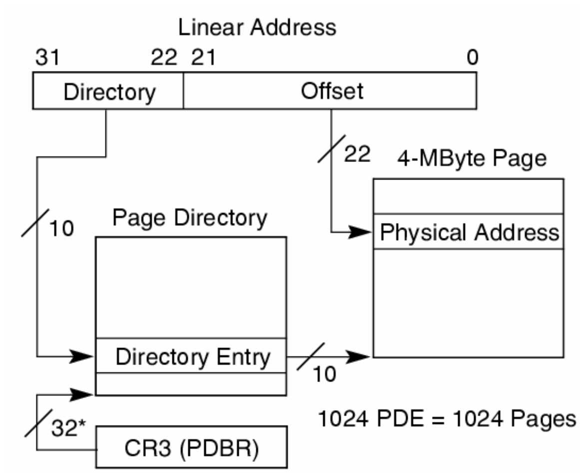
How P6 Page Tables Map Virtual Addresses to Physical Ones



4Mbyte PDE's

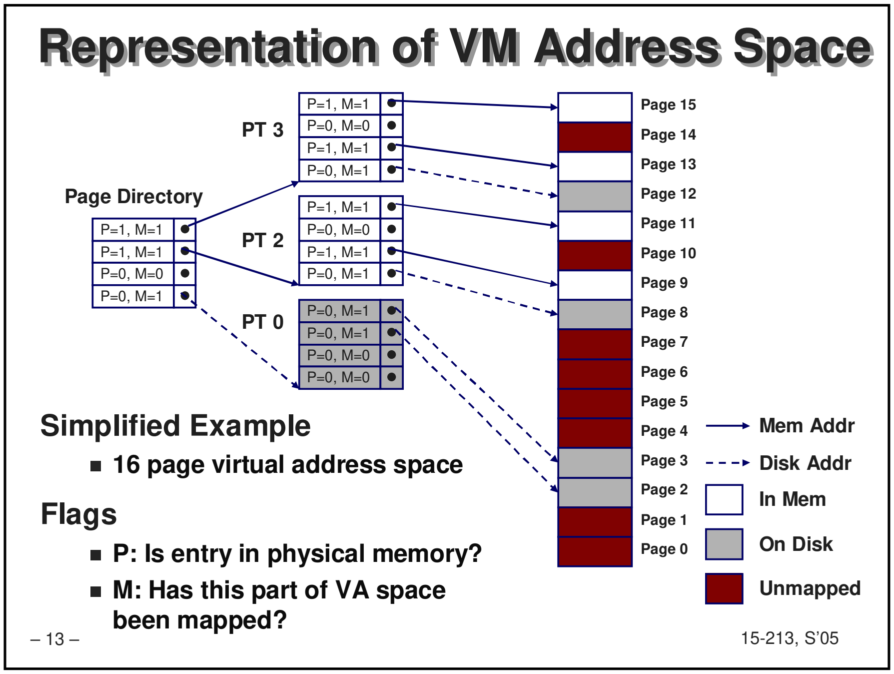


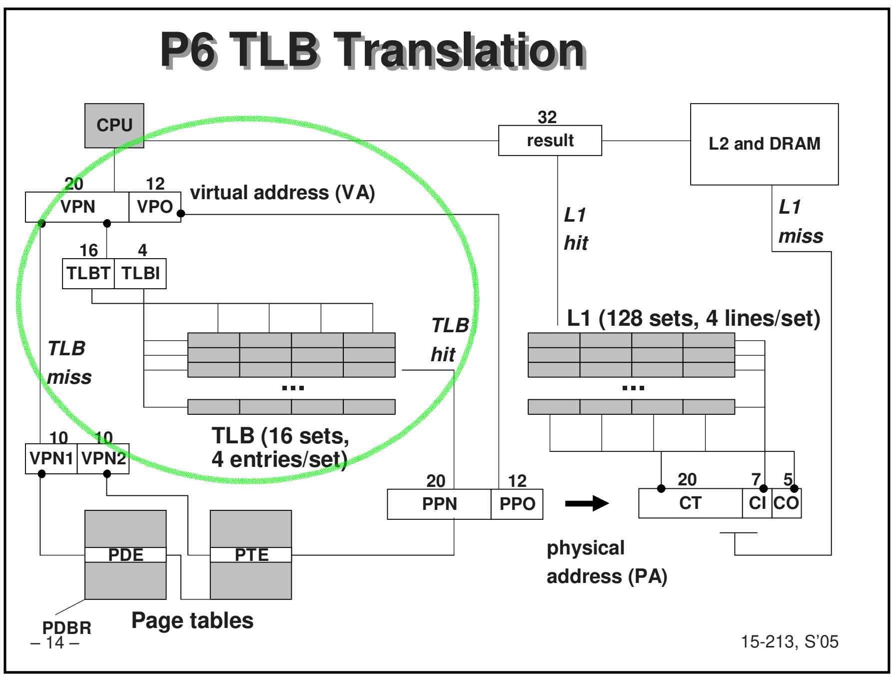
Support for 4Mbyte Pages



*32 bits aligned onto a 4-KByte boundary.

Page Directory occupies single 4KB page **(1024 4-byte entries)**


Representation of VM Address Space
Page Directory
P=1, M=1
P=1, M=1
P=0, M=0
P=0, M=1
PT 3
P=1, M=1
P=0, M=0
P=1, M=1
P=0, M=1
PT 2
P=1, M=1
P=0, M=0
P=1, M=1
P=0, M=1
PT 0
P=0, M=1
P=0, M=1
P=0, M=0
P=0, M=0
Page 15
Page 14
Page 13
Page 12
Page 11
Page 10
Page 9
Page 8
Page 7
Page 6
Page 5
Page 4
Page 3
Page 2
Page 1
Page 0
Mem Addr
Disk Addr
In Mem
On Disk
Unmapped
Simplified Example
16 page virtual address space
Flags
P: Is entry in physical memory?
M: Has this part of VA space been mapped?
– 13 –
15-213, S'05


P6 TLB Translation
CPU
32
result
L2 and DRAM
20
VPN
12
VPO
virtual address (VA)
L1
hit
L1
miss
16
TLBT
4
TLBI
TLB
hit
L1 (128 sets, 4 lines/set)
TLB
miss
...
...
10
VPN1
10
VPN2
TLB (16 sets, 4 entries/set)
20
PPN
12
PPO
20
CT
7
CI
5
CO
PDE
PTE
physical address (PA)
PDBR
Page tables
– 14 –
15-213, S'05

P6 TLB

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

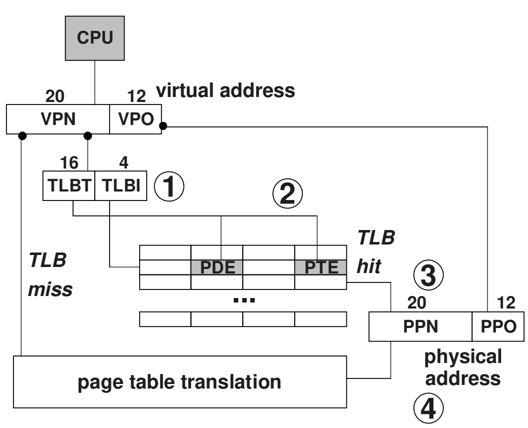
32	16	1	1
PDE/PTE	Tag	PD	V

- 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

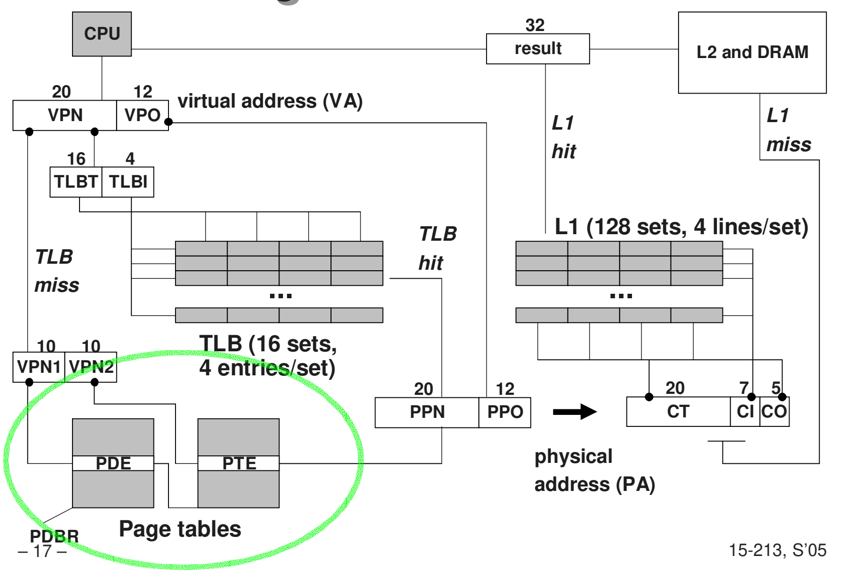
entry	entry	entry	entry	set 0
entry	entry	entry	entry	set 1
entry	entry	entry	entry	set 2
...				
entry	entry	entry	entry	set 15

Translating with the P6 TLB

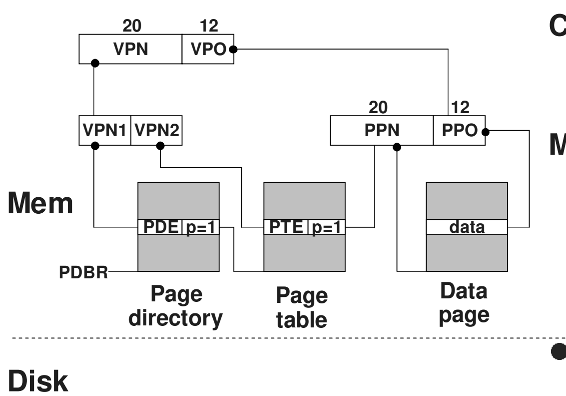


1. Partition VPN into TLBT and TLBI.
2. Is the PTE for VPN cached in set TLBI?
3. Yes: then build physical address.
4. No: then read PTE (and PDE if not cached) from memory and build physical address.

P6 Page Table Translation



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

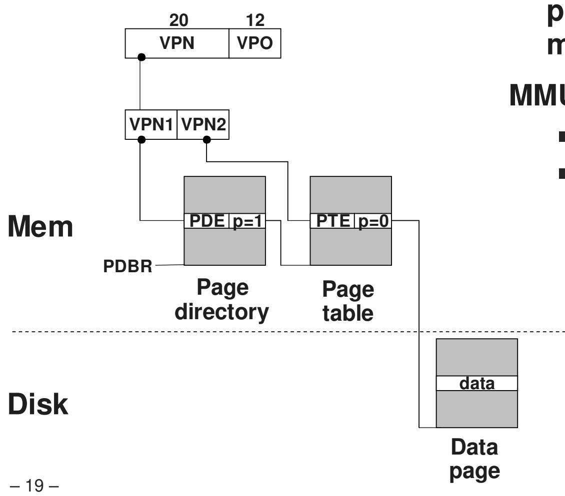


Case 1/1: page table and page present.

MMU Action:
- MMU builds physical address and fetches data word.

- OS action
  - none

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

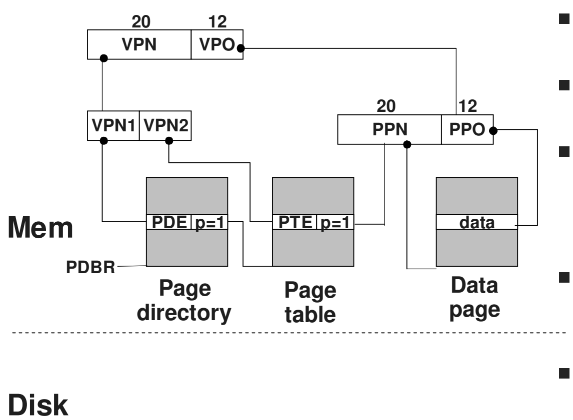


Case 1/0: page table present but page missing.

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

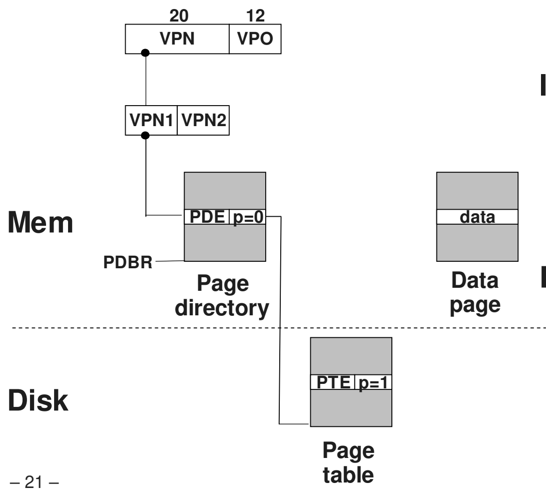
Translating with the P6 Page Tables (case 1/0, cont)



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.

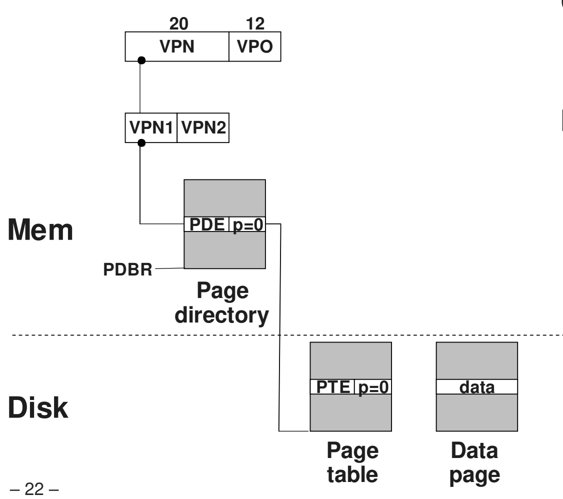
- potentially every page out requires update of disk page table.

Linux disallows this

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

– 21 – 15-213, S'05

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



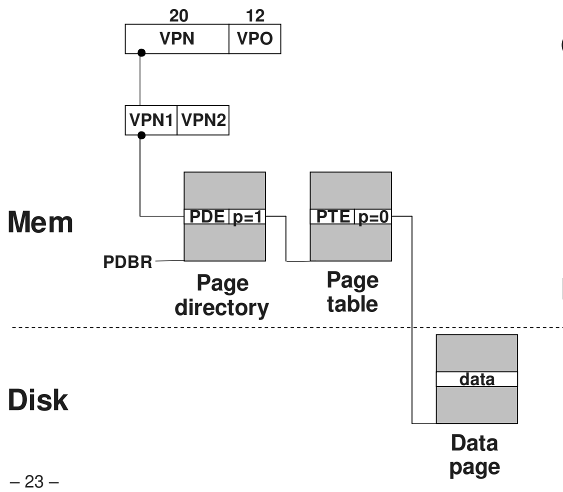
Case 0/0: page table and page missing.

MMU Action:

- page fault exception

– 22 – 15-213, S'05

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

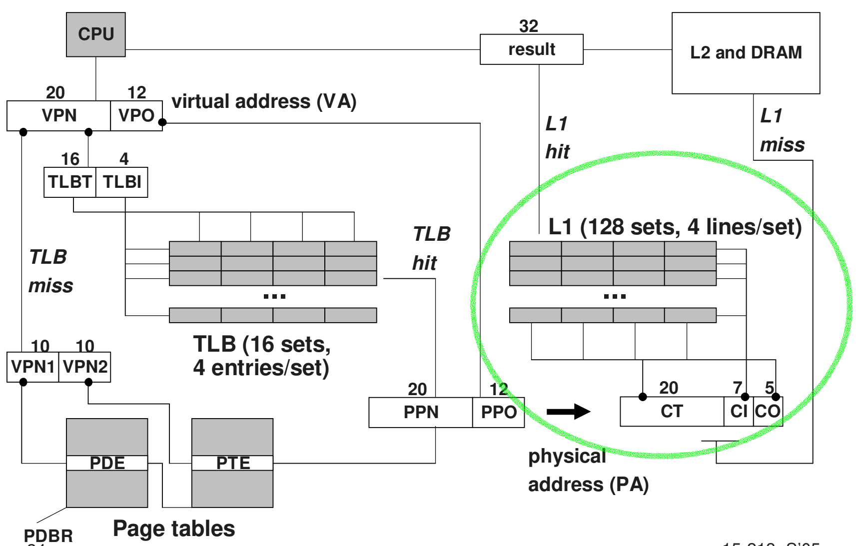


OS action:

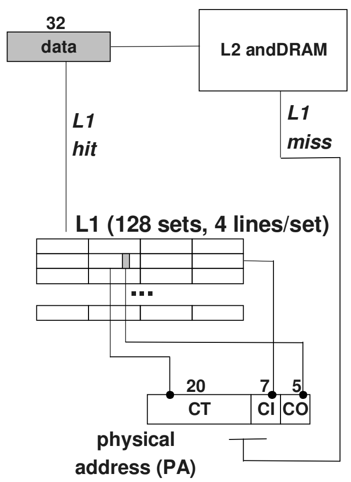
- swap in page table.
- restart faulting instruction by returning from handler.

Like case 0/1 from here on.

P6 L1 Cache Access



L1 Cache Access



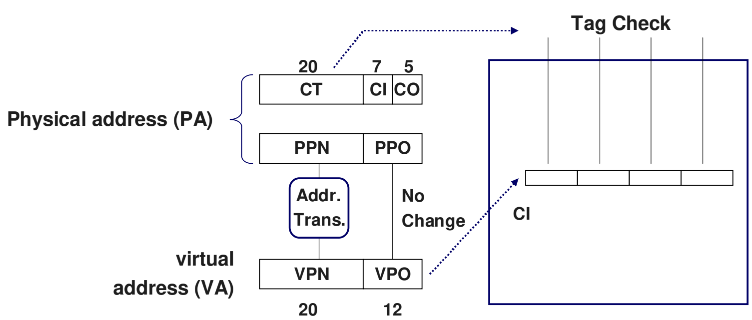
Partition physical address into CO, CI, and CT.

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

x86-64 Paging

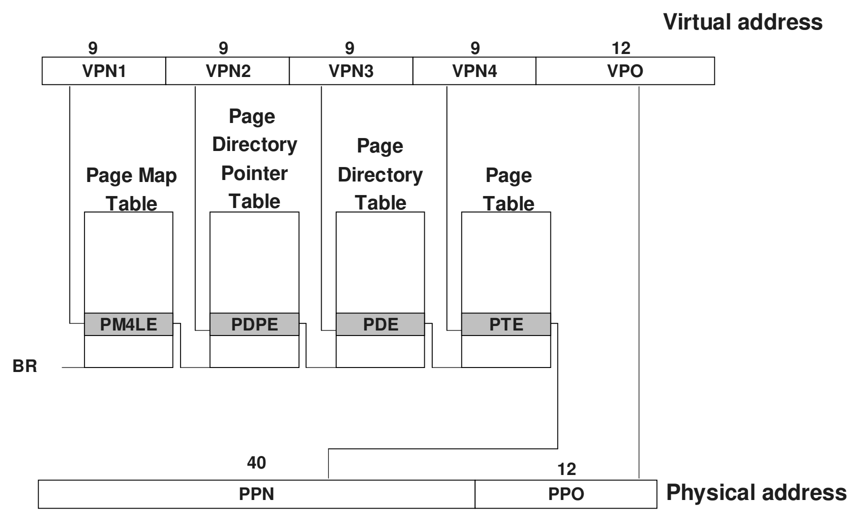
Origin

- AMD's way of extending x86 to 64-bit instruction set
- Intel has followed with "EM64T"

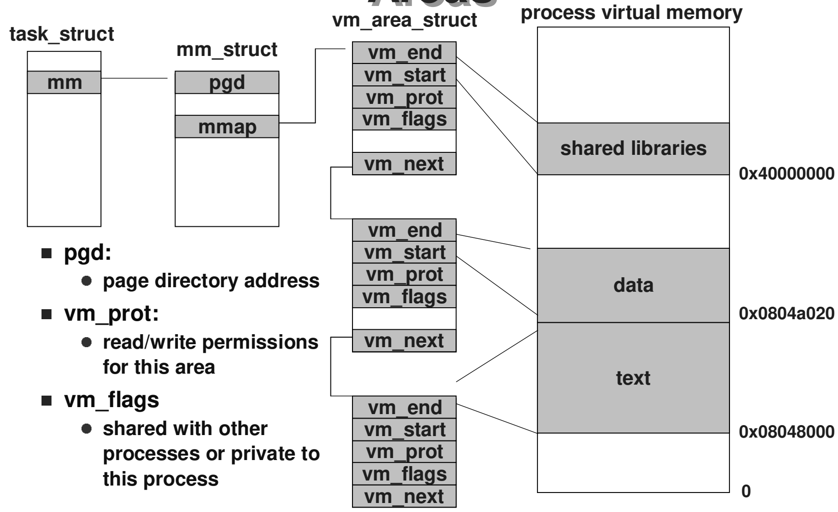
Requirements

- 48 bit virtual address
  - 256 terabytes (TB)
  - Not yet ready for full 64 bits
- 52 bit physical address
  - Requires 64-bit table entries
- 4KB page size
  - Only 512 entries per page

x86-64 Paging

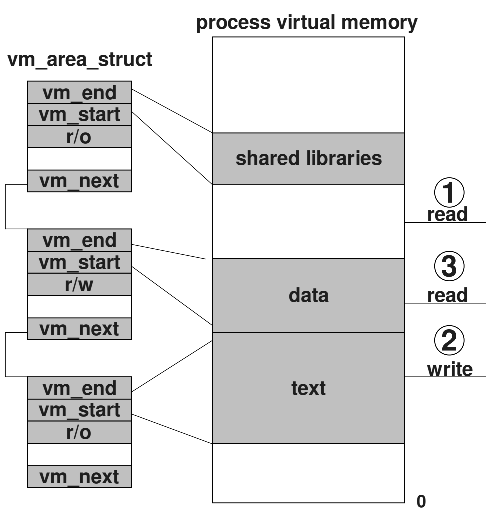


Linux Organizes VM as Collection of "Areas"



- **pgd:**
  - page directory address
- **vm_prot:**
  - read/write permissions for this area
- **vm_flags**
  - shared with other processes or private to this process

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

User-Level Memory Mapping

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

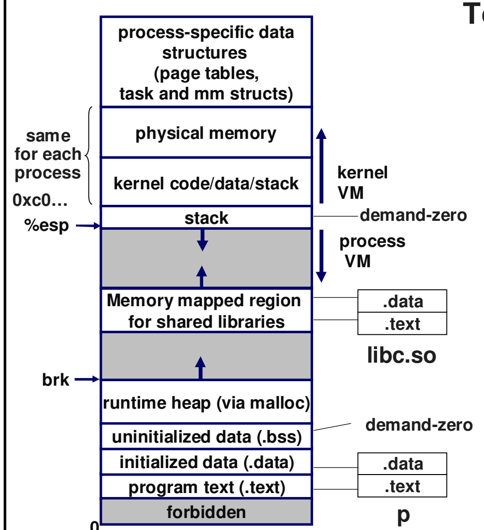
- map `len` bytes starting at offset `offset` of the file specified by file description `fd`, preferably at address `start` (usually 0 for don't care).
  - `prot`: MAP_READ, MAP_WRITE
  - `flags`: MAP_PRIVATE, MAP_SHARED
- return a pointer to the mapped area.
- Example: fast file copy
  - useful for applications like Web servers that need to quickly copy files.
  - `mmap` allows file transfers without copying into user space.

mmap() Example: Fast File Copy

```
#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

How does OS know when a page can be deallocated?

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 (mostly) 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)