

15-213

"The course that gives CMU its Zip!"

P6/Linux Memory System November 1, 2001

Topics

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

class20.ppt

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

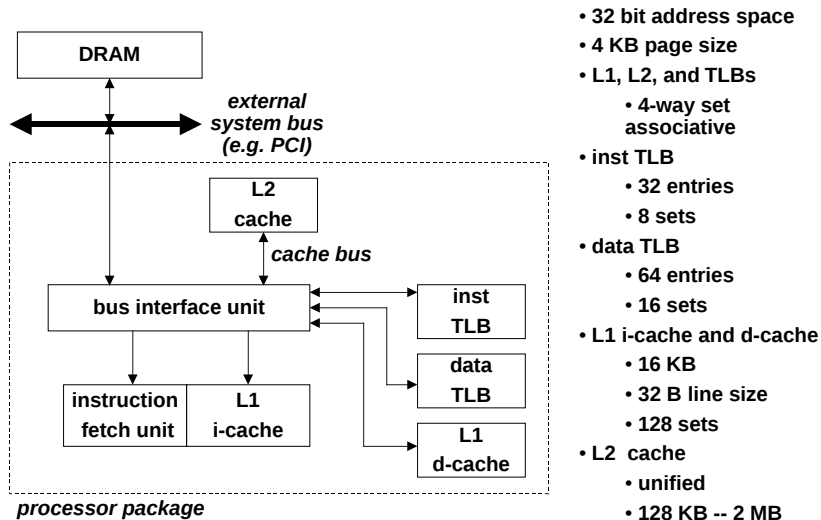
- PentiumPro (1996)
- Pentium II (1997)
 - Incorporated MMX instructions
 - » special instructions for parallel processing
 - L2 cache on same chip
- Pentium III (1999)
 - Incorporated Streaming SIMD Extensions
 - » More instructions for parallel processing

class20.ppt

– 2 –

CS 213 F'01

P6 memory system



class20.ppt

– 3 –

CS 213 F'01

Review of abbreviations

Symbols:

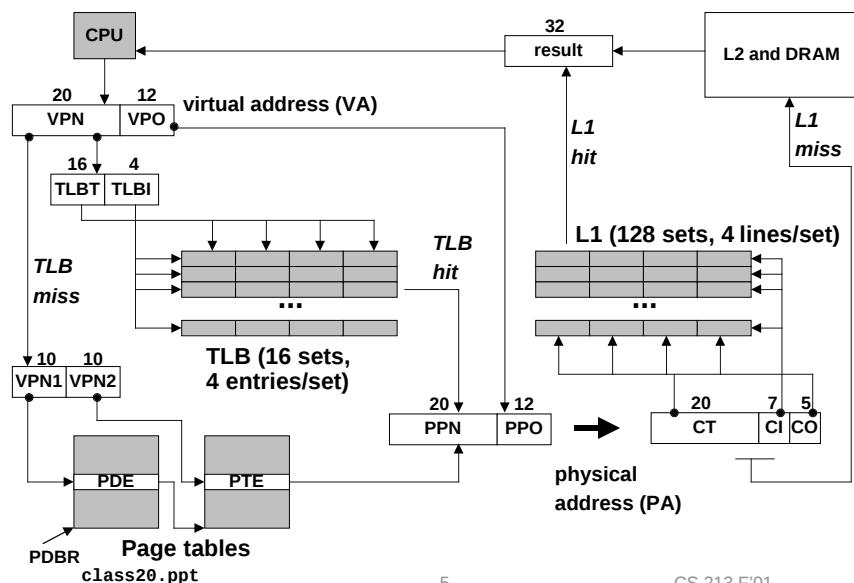
- Components of the virtual address (VA)
 - TLBI: TLB index
 - TLBT: TLB tag
 - VPO: virtual page offset
 - VPN: virtual page number
- Components of the physical address (PA)
 - PPO: physical page offset (same as VPO)
 - PPN: physical page number
 - CO: byte offset within cache line
 - CI: cache index
 - CT: cache tag

class20.ppt

– 4 –

CS 213 F'01

Overview of P6 address translation



- 5 -

CS 213 F'01

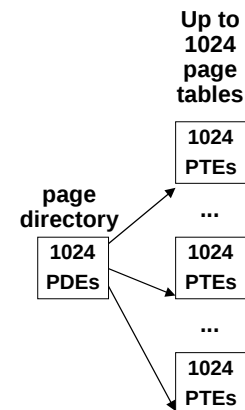
P6 2-level page table structure

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.



class20.ppt

- 6 -

CS 213 F'01

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: 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	

class20.ppt

- 7 -

CS 213 F'01

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=0		

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)

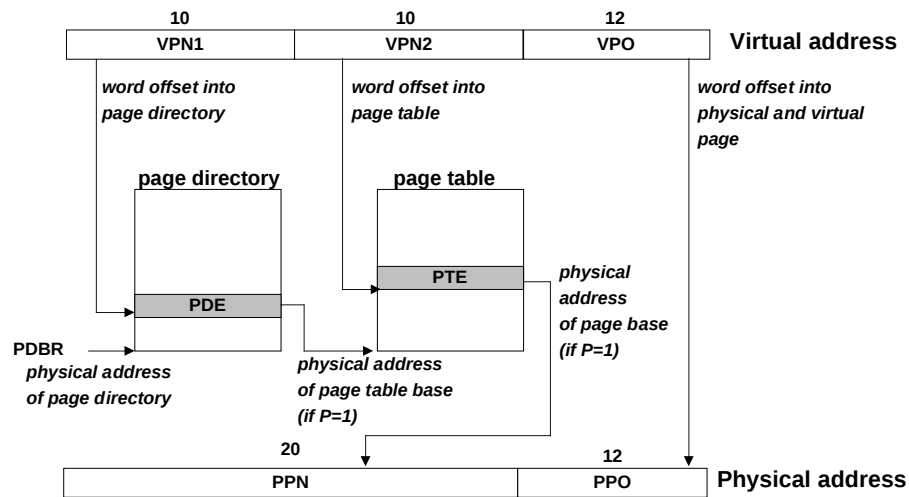
31	1	0
Available for OS (page location in secondary storage)	P=0	

class20.ppt

- 8 -

CS 213 F'01

How P6 page tables map virtual addresses to physical ones

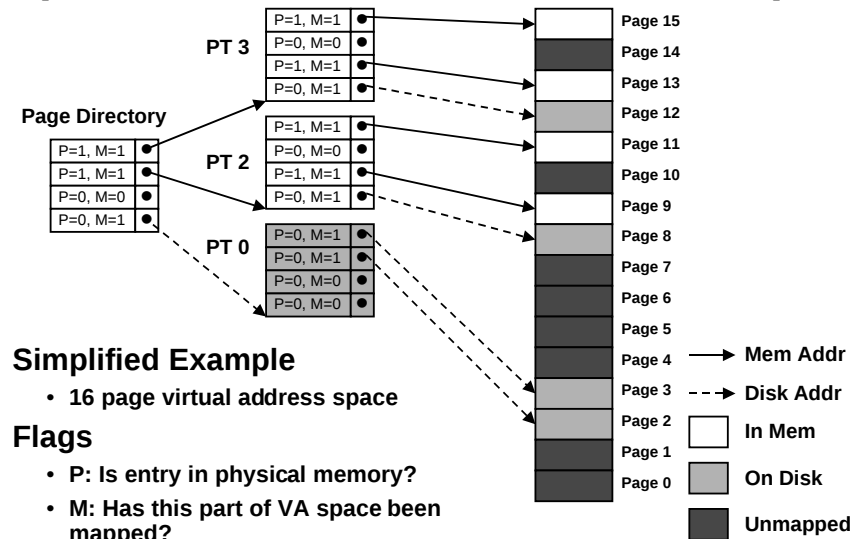


class20.ppt

- 9 -

CS 213 F'01

Representation of Virtual Address Space



Simplified Example

- 16 page virtual address space

Flags

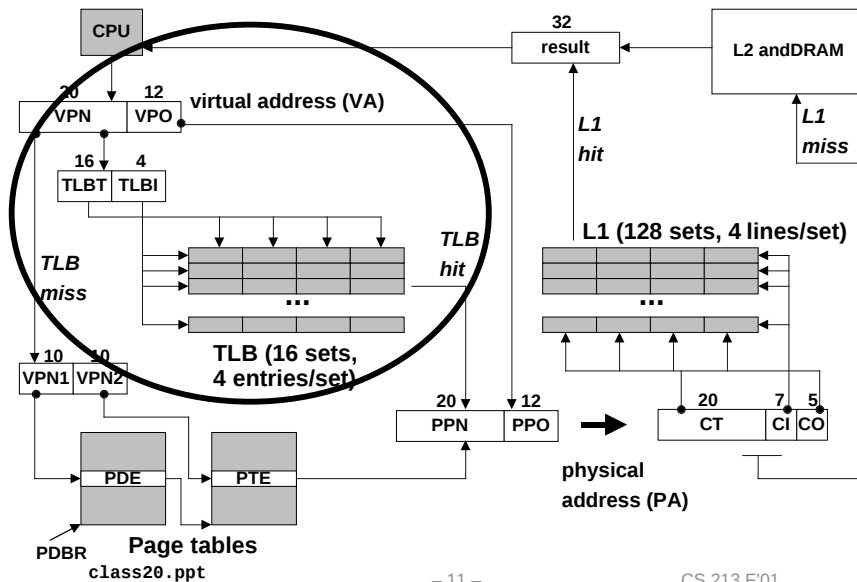
- P: Is entry in physical memory?
- M: Has this part of VA space been mapped?

class20.ppt

- 10 -

CS 213 F'01

P6 TLB translation



class20.ppt

- 11 -

CS 213 F'01

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

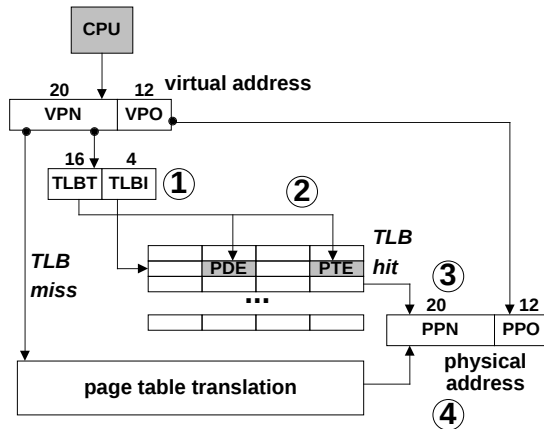
entry	entry	entry	entry	set 0
entry	entry	entry	entry	set 1
entry	entry	entry	entry	set 2
...	...	...	...	...
entry	entry	entry	entry	set 15

class20.ppt

- 12 -

CS 213 F'01

Translating with the P6 TLB

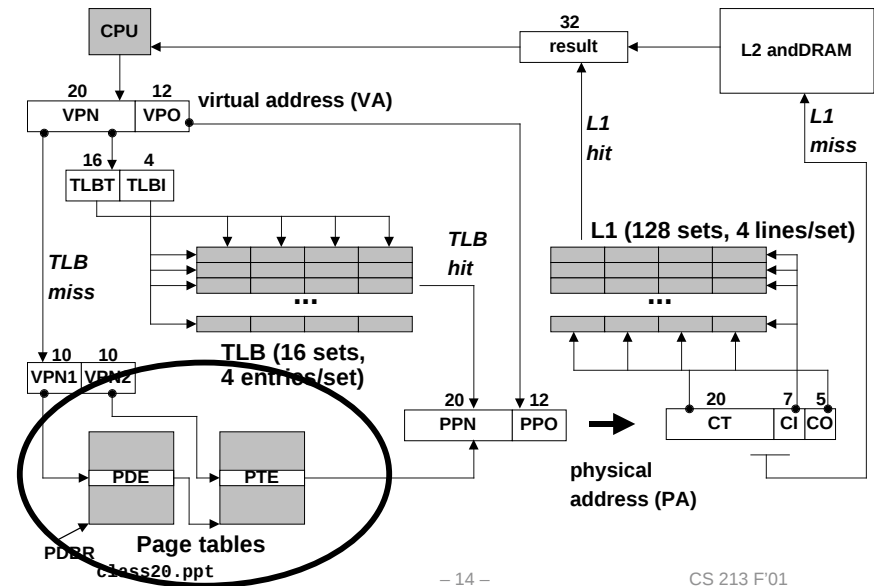


class20.ppt

- 13 -

CS 213 F'01

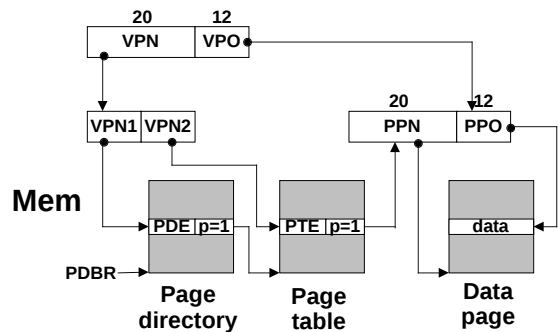
P6 page table translation



- 14 -

CS 213 F'01

Translating with the P6 page tables (case 1/1)



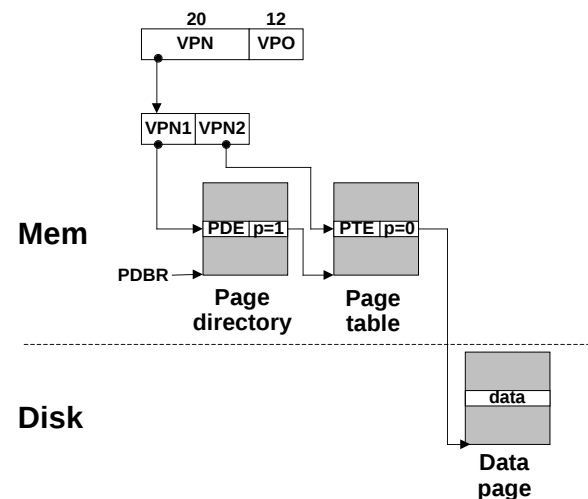
Disk

class20.ppt

- 15 -

CS 213 F'01

Translating with the P6 page tables (case 1/0)



class20.ppt

- 16 -

CS 213 F'01

Case 1/1: page table and page present.

MMU Action:

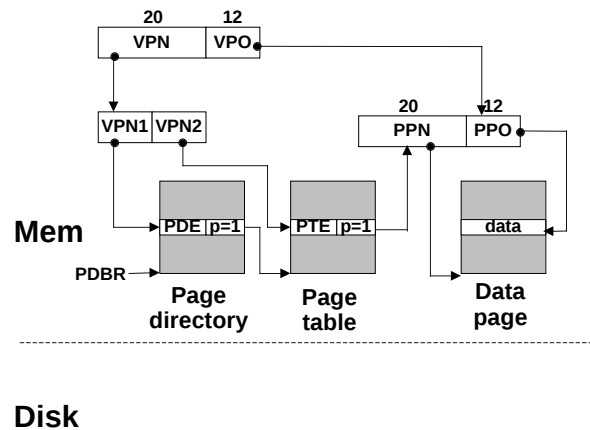
- MMU build physical address and fetch data word.
- OS action
- none

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)



class20.ppt

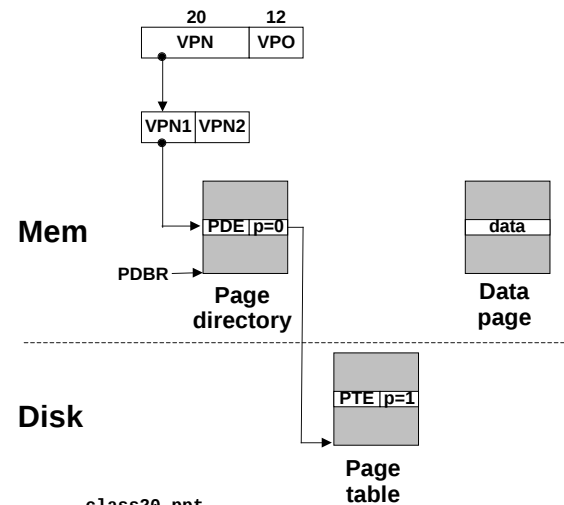
- 17 -

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.

CS 213 F'01

Translating with the P6 page tables (case 0/1)



class20.ppt

- 18 -

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

Introduces consistency issue.

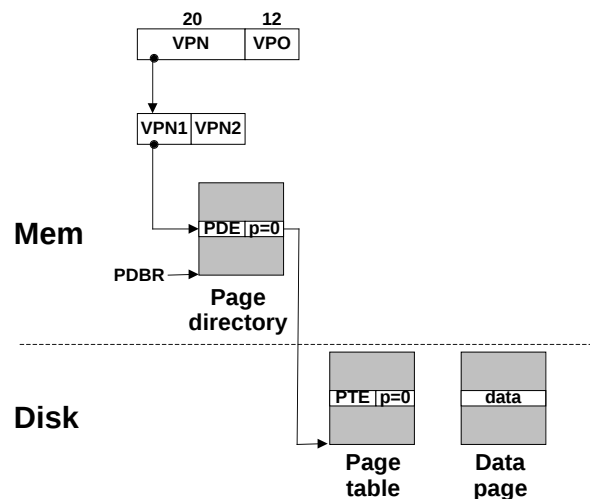
- potentially every page out requires update of disk page table.

Linux disallows this

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

CS 213 F'01

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



class20.ppt

- 19 -

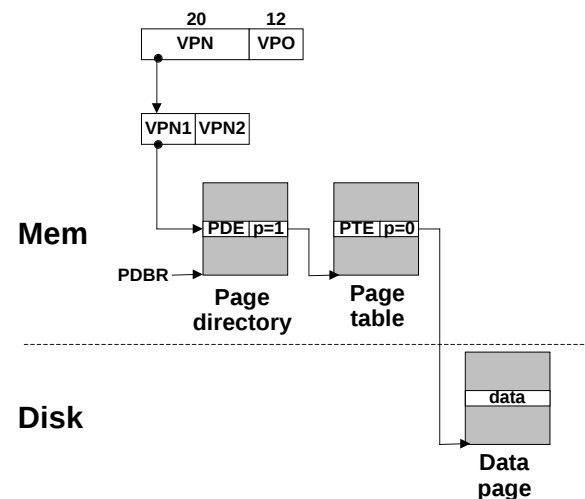
Case 0/0: page table and page missing.

MMU Action:

- page fault exception

CS 213 F'01

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



class20.ppt

- 20 -

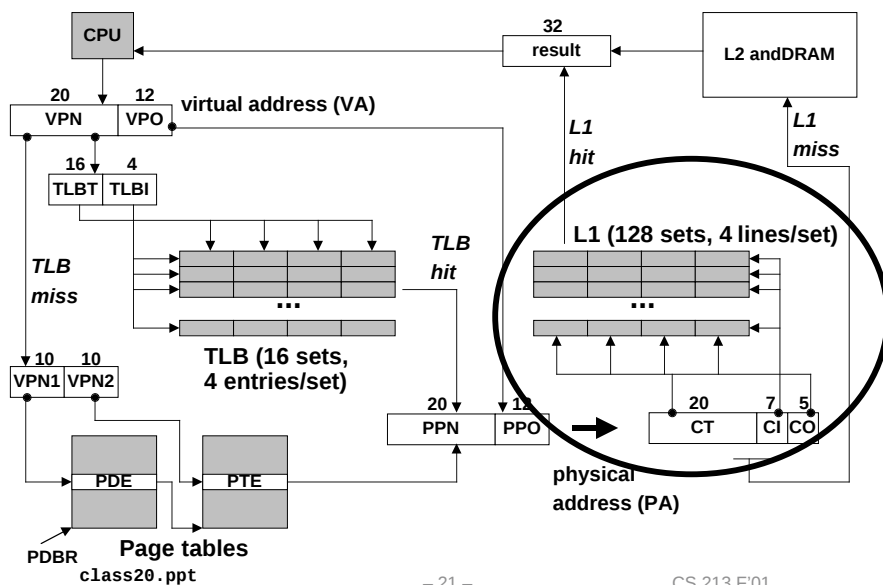
OS action:

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

Like case 0/1 from here on.

CS 213 F'01

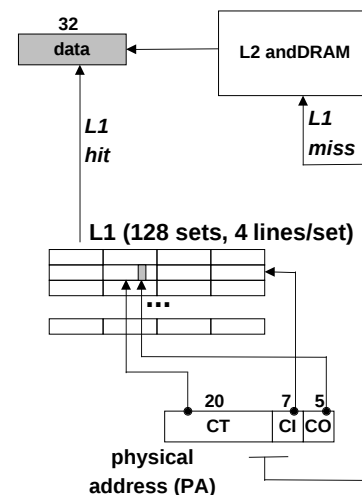
P6 L1 cache access



- 21 -

CS 213 F'01

L1 cache access



- 22 -

CS 213 F'01

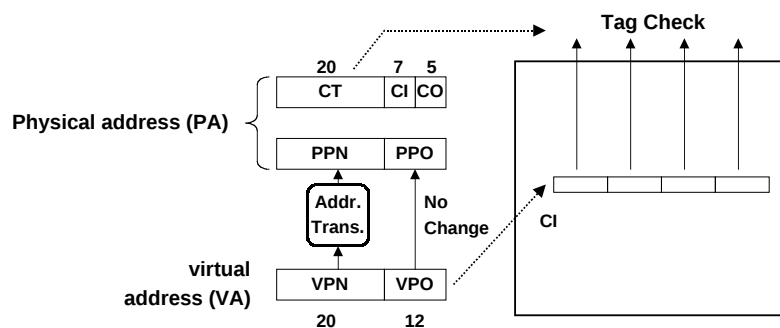
Partition physical address into CO, CI, and CT.

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



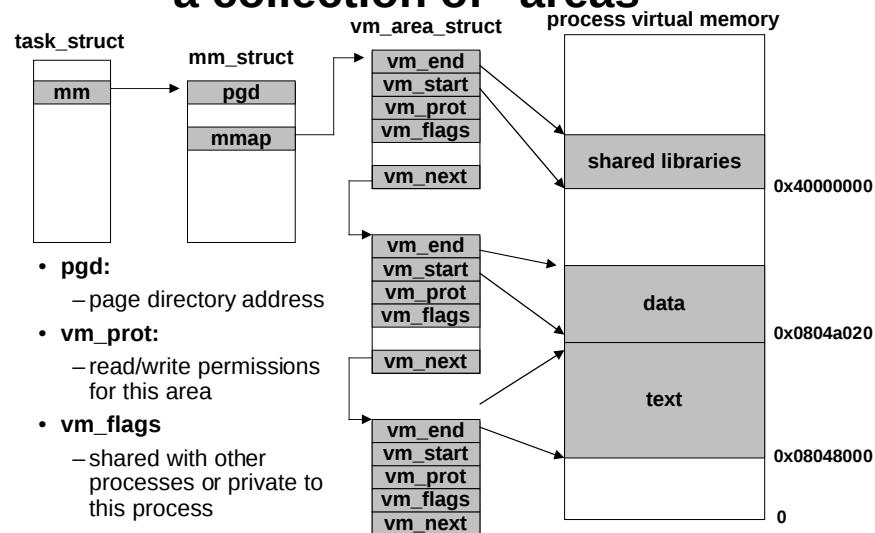
- 23 -

CS 213 F'01

Observation

- Bits that determine CI identical in virtual and physical address
- Can index into cache while address translation being performed
- Then check with CT from physical address
- "Virtually indexed, physically tagged"

Linux organizes VM as a collection of "areas"

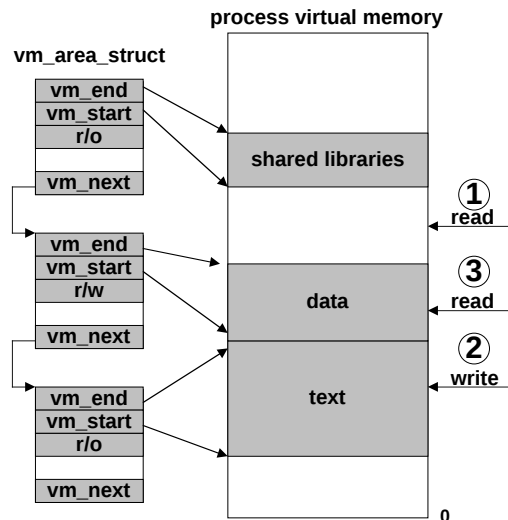


- 24 -

CS 213 F'01

- **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)

class20.ppt

- 25 -

CS 213 F'01

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

class20.ppt

- 26 -

CS 213 F'01

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.

class20.ppt

- 27 -

CS 213 F'01

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 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);

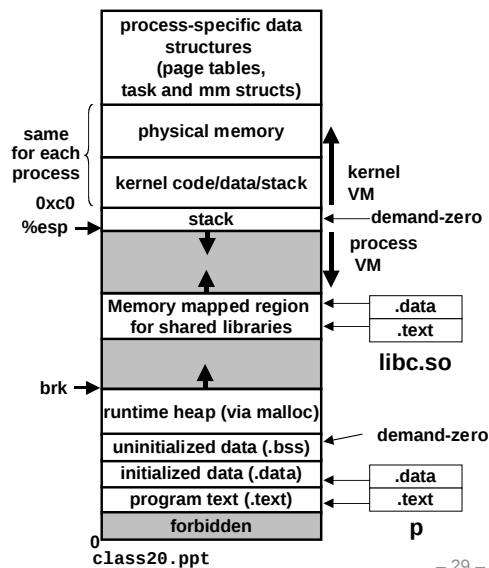
/* write the VM area to stdout */
write(1, bufp, size);
}
```

class20.ppt

- 28 -

CS 213 F'01

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

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)