Linux* Virtual Memory Management

Intel® China Software Center

SSG Core Software Division

Physical Memory

- Linux* divide physical memory in to the data structure of node. Each node represent a bank of memory attached to a NUMA(Non Uniform Memory Access) machine. On SMP(Symmetric Multi Processor) and UP(Single CPU) system, there is only 1 node.
- Each node is divided into zones, There are currently 4 zones in Linux*, ZONE_DMA, ZONE_DMA32, ZONE_NORMAL, ZONE_HIGHMEM.
- Zone is the core structure of page allocator.

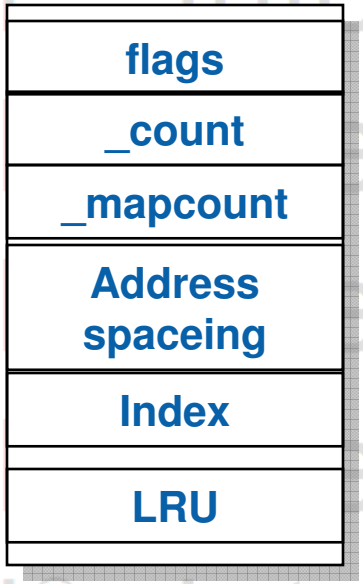
Physical Memory

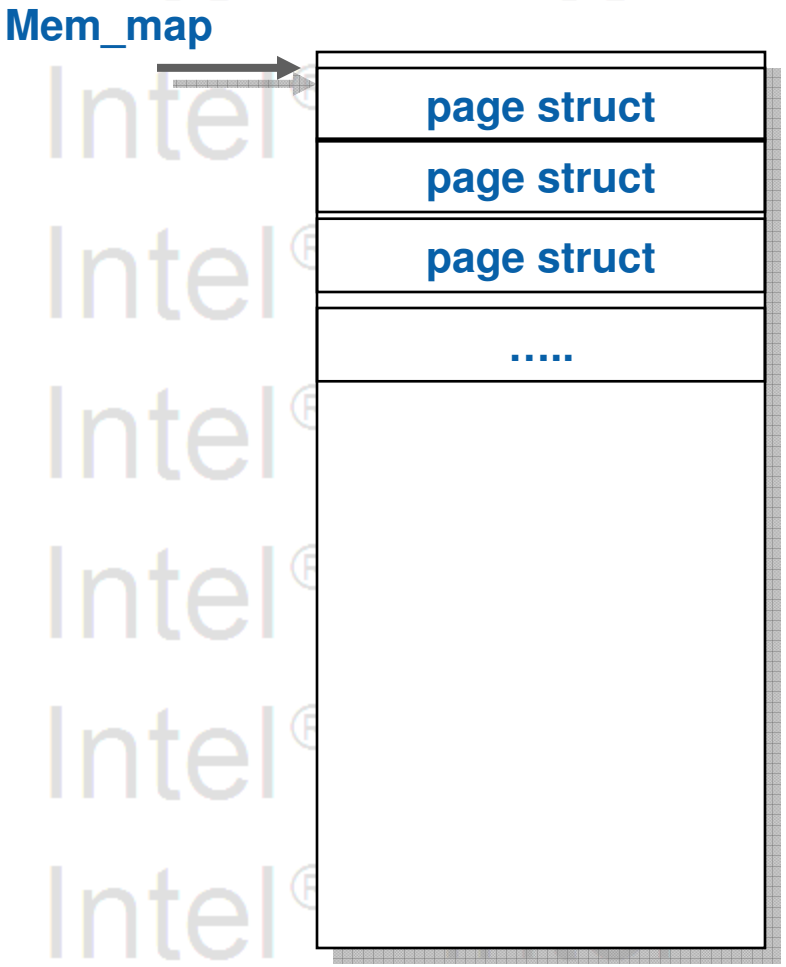
- Zone may contains holes, there is no assumption that physical memory in a zone is continuous.
- Linux* kernel will setup node and zone structure at the time of booting according to the information passed from BIOS. On i386, memory range information is usually stored in E820 table and EFI memory mao.

Page structure

- Linux* assume memory is based on unit of page
- Each physical page of memory is described with a PFN (Page Frame Number).
- There is a structure page represent each physical page
- Page structure is the basic unit that kernel handle with memory allocation and operation.
- Page structure itself is saved in a per node structure mem_map
- Kernel will initialize all the page structure at boot time.
- Pages are often organized in list

Page structure





Virtual Memory

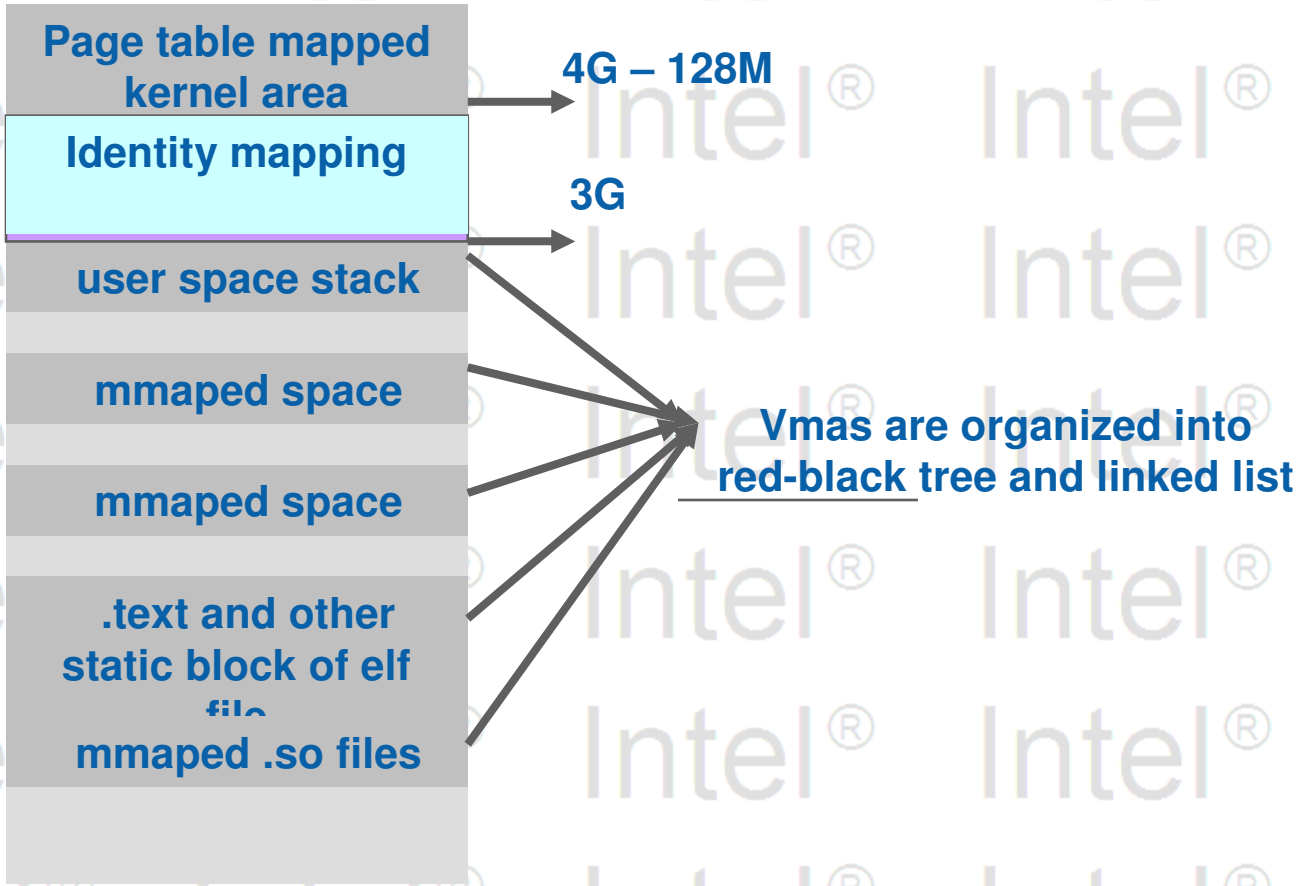
- Virtual memory is a concept that each process can has his own linear memory space.
- There are hardware unit which can translate virtual address into physical address called MMU (memory management unit). The basic structure of MMU is page table.
- Each process may have it's own virtual memory space.
- In Linux*, kernel and user space are in the unified virtual memory space.
- On i386 platform, virtual memory space is divide into 2 regions, 0-3G for user space, 3G-4G for kernel space
- User space program can allocate virtual memory spaces via system calls mmap munmap and brk.

Virtual Memory

- Kernel can provide more virtual memory space than physical memory by a method called demand paging.
- On i386, Kernel can use more physical than 4G although 1 process's VM space is only 4G.
- There is a mm structure in task structure to represent a tasks virtual memory space status. Different threads within a process share 1 mm structure, kernel thread set the mm structure to NULL.
- There is a red-black tree of vm_struct vma list in mm structure, those vma are also organized into a sorted list.
- vm_strcuture has different flags, kernel use this flag to support protection control of virtual memory.

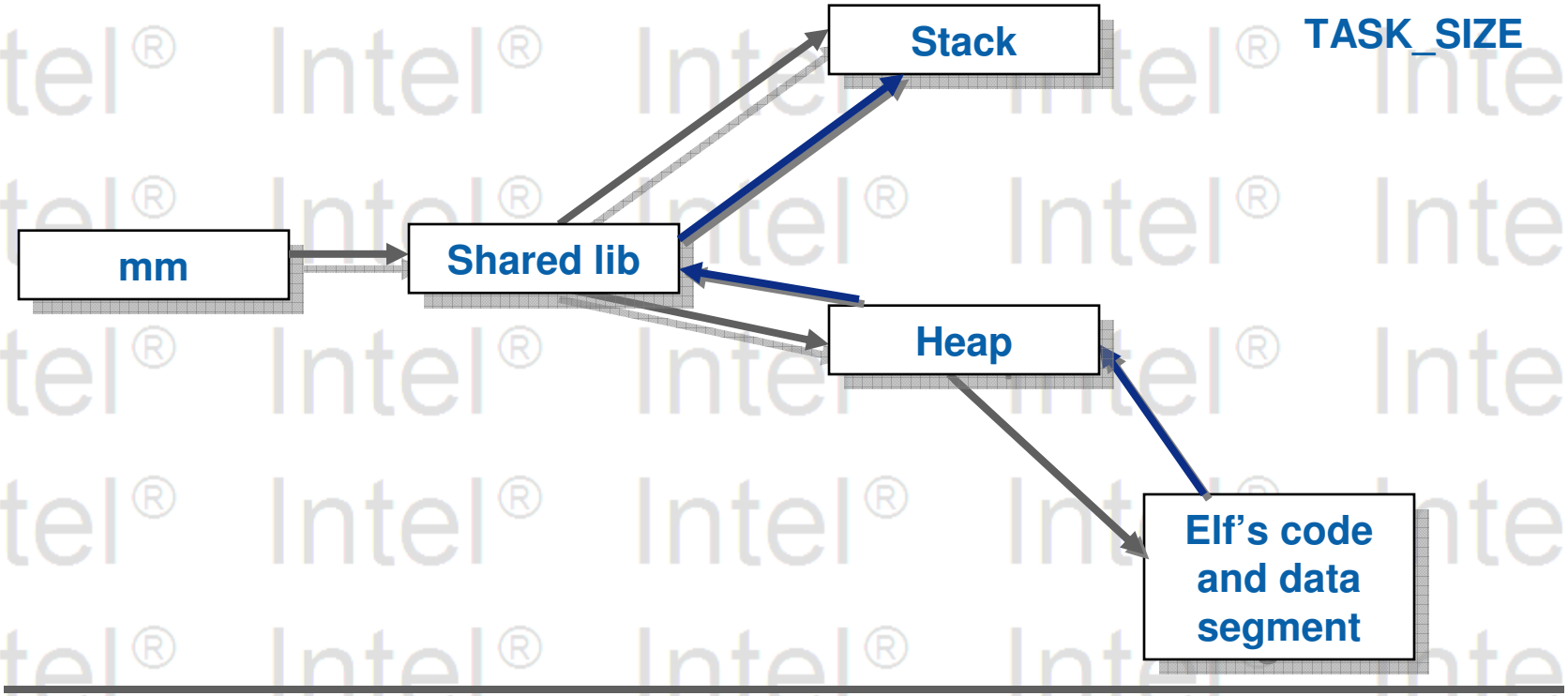
Virtual Memory

- typical virtual memory mapping of a on I386



VMA

- Vma structure is mainly used in userspace for define a processes virtual address ranges.
  define the access right of arrange of addresses.



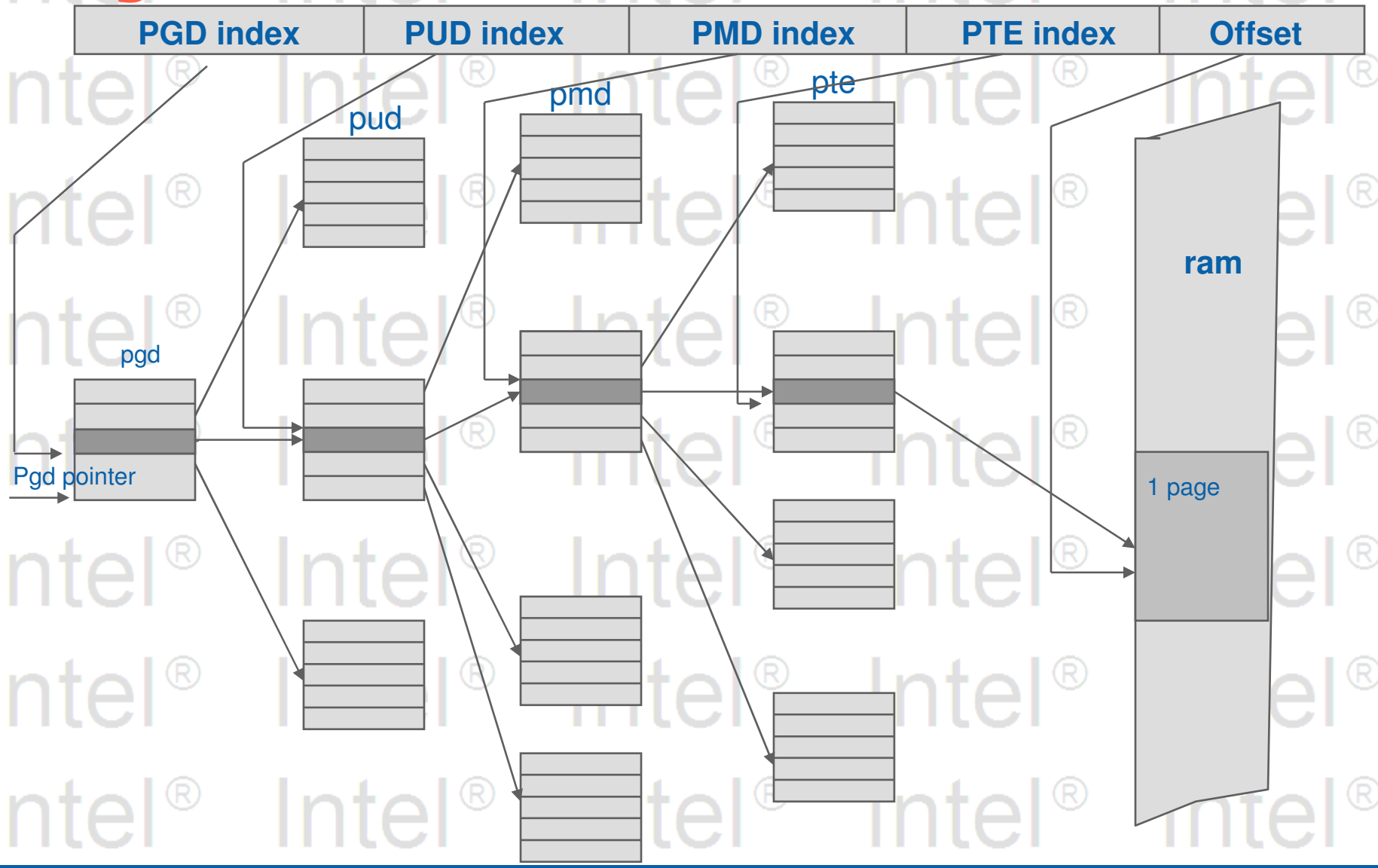
Virtual Memory

- Page tables is the primary structure that kernel used to map virtual address to physical address
- Page tables is a multi-way multi-level tree structure.
- Different architecture may support different level of page tables.
- Kernel has generic code and macro to support 4 level page table, page table may be collapsed by macro define.
- Every mm structure has a pointer pgd pointed to base level entry of page tables. On i386, this pgd is corresponding to CR3 register.
- IA32 use 2 levels page table and 3 levesl page table

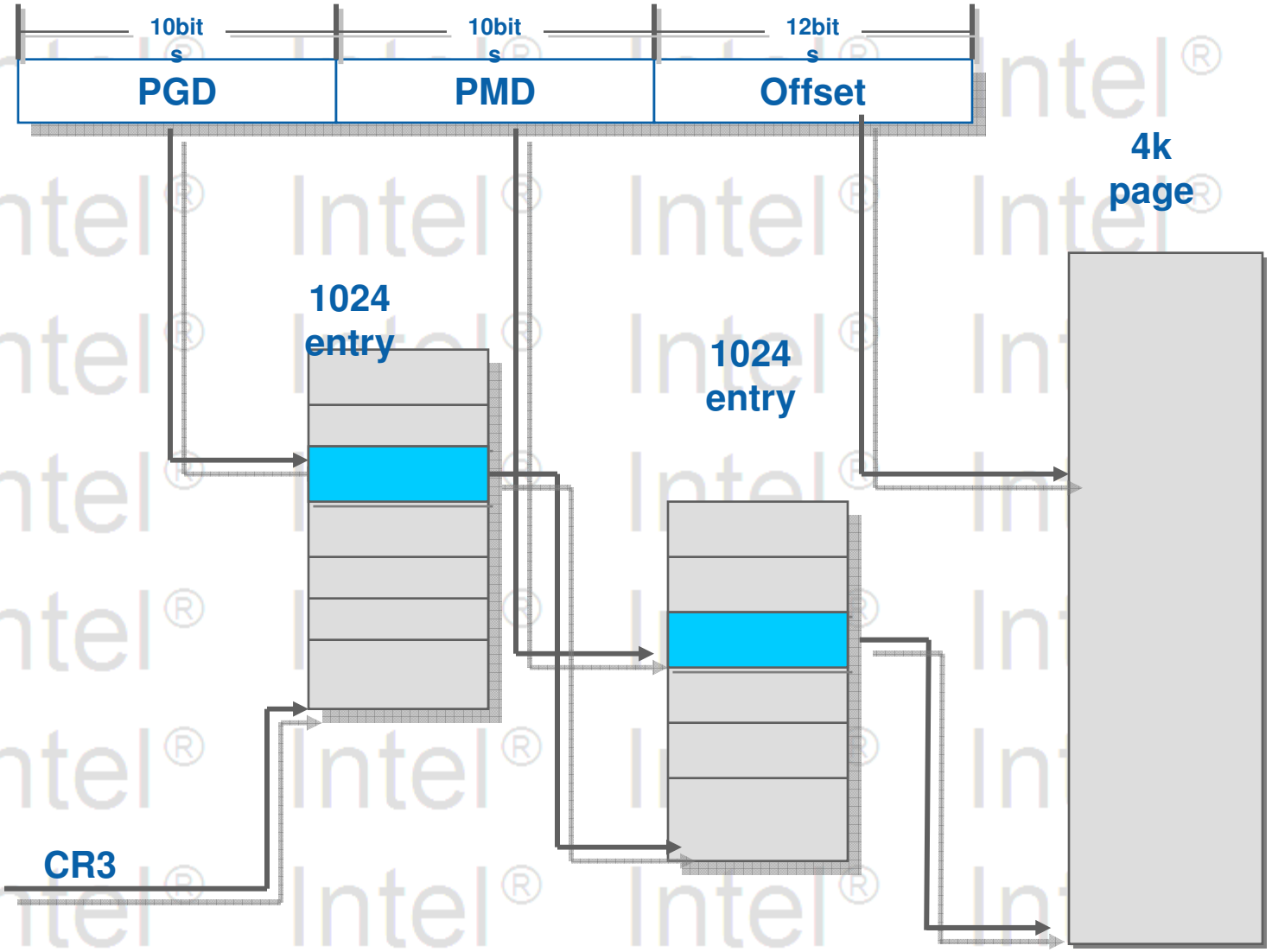
Summary

- Concept of physical memory
- Concept of virtual memory
- Page structure
- Reference code mm/mmap.c, mm/page_alloc.c, mm/bootmem.c arch/i386/mm/mmap.c,

Page table

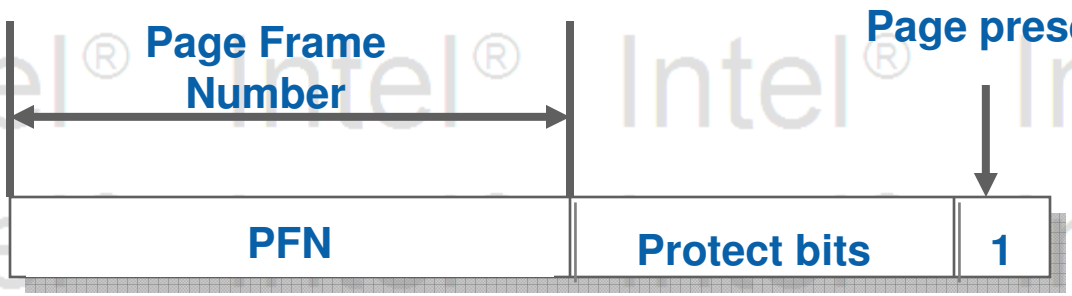

PGD index
PUD index
PMD index
PTE index
Offset
pgd
pud
pmd
pte
ram
Pgd pointer
1 page

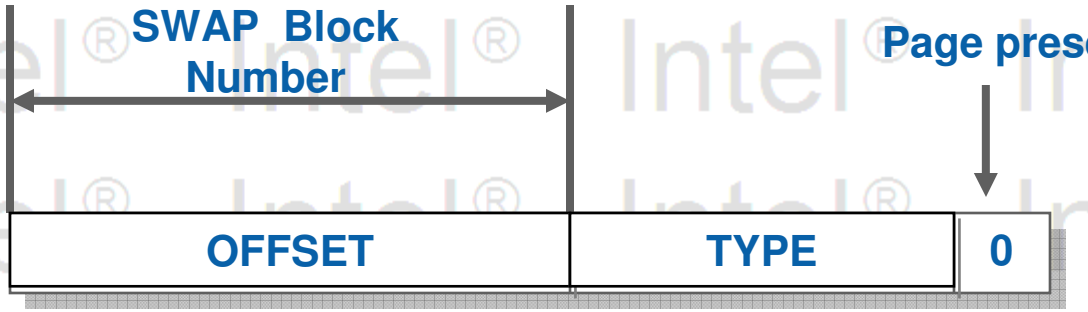
I386 2-LEVEL page table



Virtual Memory

- PTE entry





Virtual Memory

- TLB (Translation Look Aside buffer)

  TLB is a kind of associative cache which is used to translated virtual address to physical address.

  On I386, if hardware can't find a TLB entry , it will automatically issue a search on page table. If page table entry is found, it will insert a TLB entry, otherwise hardware will inject a page fault where OS can handle.

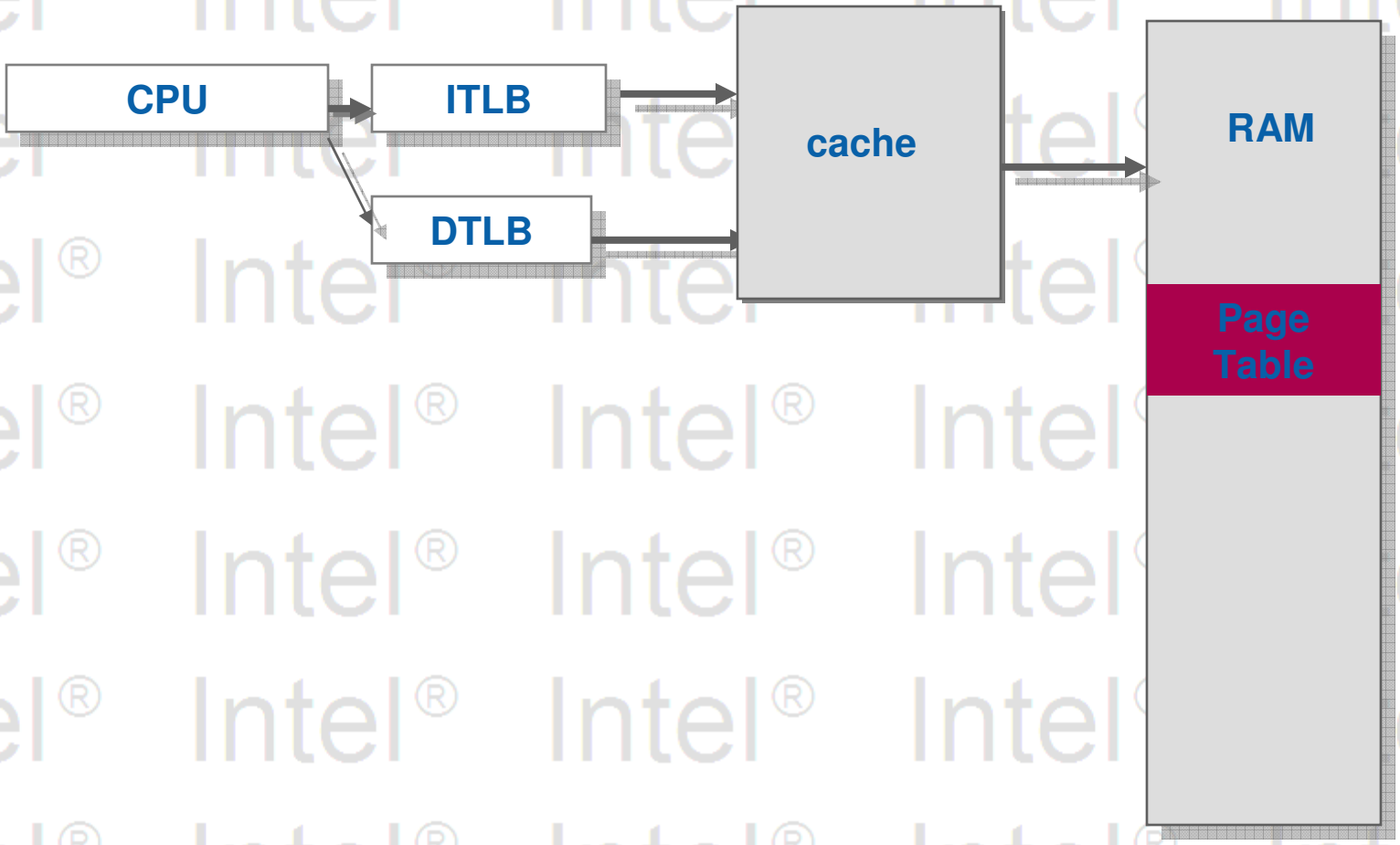
  There are 2 ways to flush TLB, write to CR3 or with invlpg.

  Some pages are mapped globally that do not need to flush at the time of task switch.

  I386 need a TLB flush at the time of task switch, and at the time of unmap a region of virtual address.

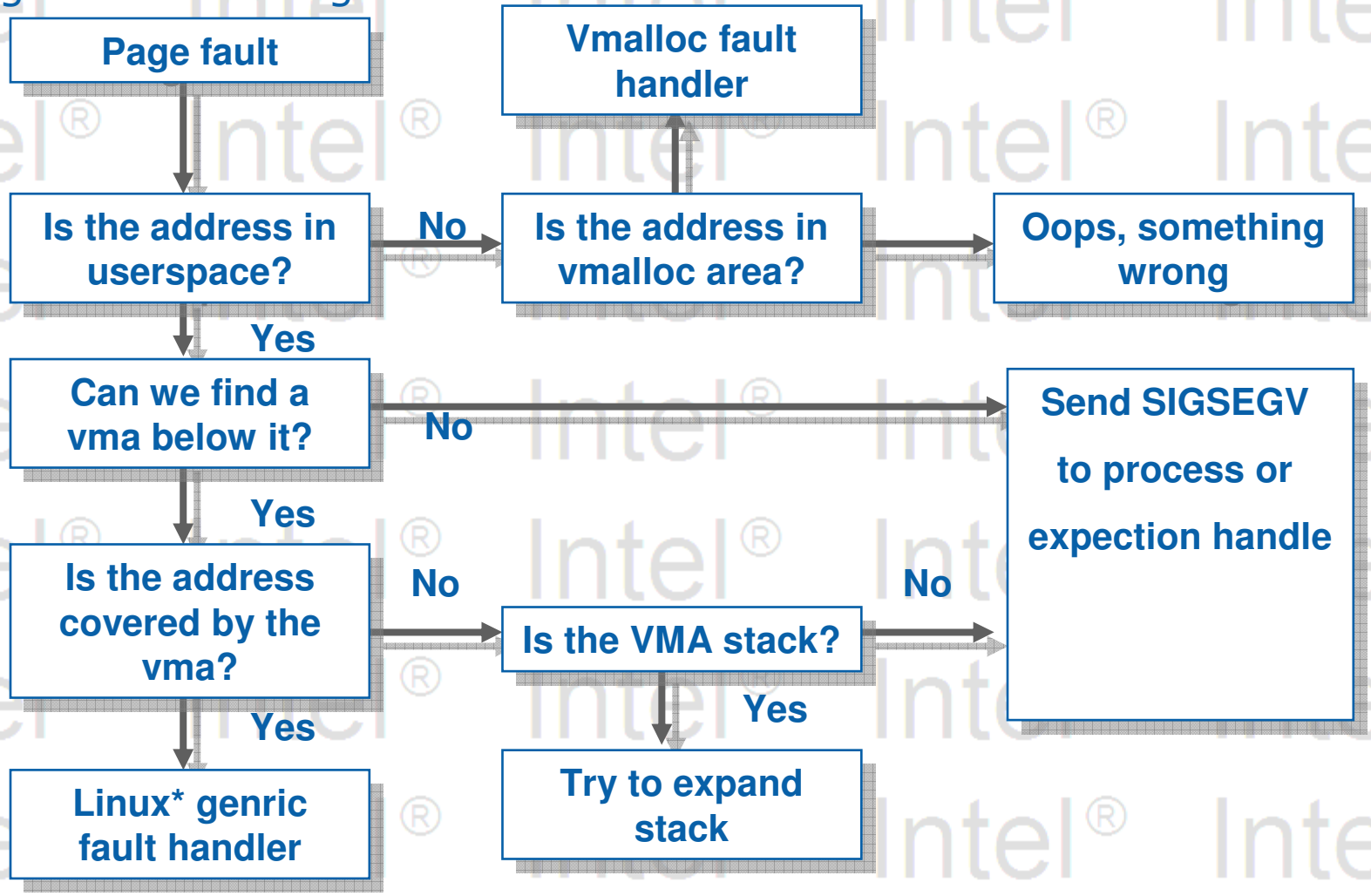
  on SMP system, tlb flush need to be broadcasted to each CPU.

CPU to



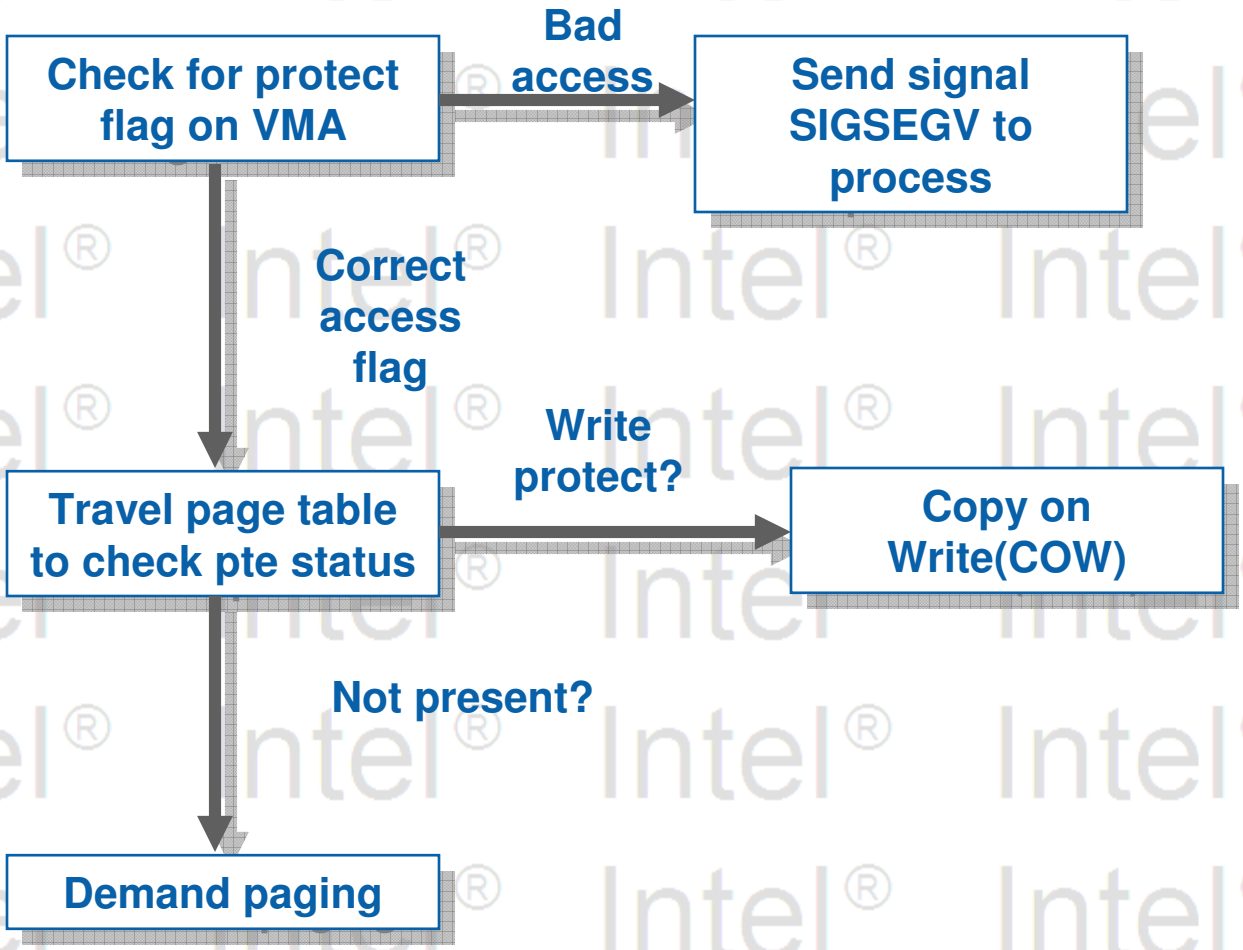
Page fault

- Page fault handling



Page fault

- Page fault handling



Virtual Memory

- Kernel mapping of physical memory

  Kernel will setup direct mapping from virtual to physical address in ZONE_NORMAL.

  The identity mapping virtual address is used inside kernel code.

  By this way, virtual to physical address translation calculation is simple.

  #define __va(x) ((void *)((unsigned long)x + PAGE_OFFSET)

  #define __pa(x) ((unsigned long)x – PAGE_OFFSET)

VMALLOC

- Kernel can also map non- contiguous physical pages into contiguous virtual pages address via vmalloc interface.
- Vmalloc virtual address is like user space virtual address

  vmalloc pages are on demand.

  different process will share same pages of vmalloc

  but they will have different page directory for 1 page
- Vmalloc address is defined above VMALLOC_START and limited by VMALLOC_END
- vmalloced address can't be used for DMA
- When there is no current mm, vmalloc may find and allocate page table entry in init_mm.
- For a vmlloced space, different processes may have different higher level page table entry but share the last level pte entry.

Hi mem support

- High memory support

  On 32bit platform like IA32, virtual address is limited. There is only 896M address can be directly mapped.

  Linux* defined interface to access to memory that can't be directly accessed via identity mapped segments.

  There are some primary interfaces.

  kmap(page) /*map page frame into virtual space */

  kunmap(page) /*unmap page frame from virtual space */

Paging init and clean up

- Paging init

  on a fork, child process will clone mm structure from his parent.

  every mm structure is cloned from init_mm

  on kernel startup, kernel will add each level pagetable entry for identity mapping to init_mm.

  on a process exit, kernel will reclaim page tables and flush tlb

Summary

- Paging concept and implement.
- Page fault handling
- Reference code mm/memory.c arch/i386/mm/fault.c

API

- Detailed macros and types fo handle and access page tables

typedef pgd_t	/* global-directory entry */
typedef pmd_t	/* L2-directory entry */
typedef pte_t	/* page table entry */
pgd_t *pgd_offset(mm, addr)	/* get pgd entry for addr */
pgd_t *pgd_k_offset(addr)	/* get kernel pgd entry for addr */
pmd_t *pmd_offset(pgd_entry, addr)	/* get pmd entry for addr */
pte_t *pte_offset(pmd_entry, addr)	/* get pte entry for addr */
int pgd_none(pgd_entry)	/* check if pgd_entry is mapped */
int pgd_present(pgd_entry)	/* check if pgd_entry is present */
int pgd_bad(pgd_entry)	/* check if pgd_entry is valid */

API

int pmd_none(pmd_entry)	/* check if pmd_entry is mapped */
int pmd_present(pmd_entry)	/* check if pmd_entry is present */
int pmd_bad(pmd_entry)	/* check if pmd_entry is valid */
int pte_none(pte)	/* check if pte is mapped */
int pte_present(pte)	/* check if pte is present */

Page table code sample

- Sample code of access a processes page table

```
void print_physical_address(struct mm_struct *mm, unsigned long addr)
{
    pgd_t  * pgd = pgd_offset(mm, addr);
   if (pgd_present(*pgd)) {
                pud_t * pud = pud_offset(pgd, addr);
                if (pud_present(*pud)) {
                                pmd_t * pmd = pmd_offset(pud, addr);
                                if (pmd_present(*pmd)) {
                                                pte_t *pte      = pte_offset(pmd, addr);
                printk("va %x%lx -> pa  %x%lx\n", va,
                                                                __pa(page_address(pte_page(*pte)));
                                }
                }
   }
    printk("no mapping found for address 0x%lx\n", addr);
}
```

Virtual Memory

- Kernel and user space data copy

  Sometimes data need to be copied from kernel space to user space or vise visa.

  Use memcpy is a bad idea.

  That is because
  - **The copy code is run on ring 0 in kernel mode, simple memory copy may corrupt or leak important data.**
  - **On some implement, user and kernel has different memory space.**

  Linux* provide interface to access and transfer data between kernel and user space.

  in asm/access.h.

Virtual Memory

User access helpers

access_ok(type, addr,size)	Check if a user space pointer is ok
put_user(x, ptr)	Write a value into user space
get_user(x, ptr)	Read a value from user space
unsigned long copy_to_user(void *to, void *from, unsigned long n)	copy n bytes to user space
unsigned long copy_from_user(void *to, void *from, unsigned long n)	copy n bytes from user space

Virtual Memory

- Exception tables

When a page fault happens and kernel can't find mapping for it.

Kernel will sent SIGSEGV or do exception handling according to privilege level.

Kernel and modules has their own exception table.

exception table is a sorted table of 2 element structure.

```
struct exception_table_entry {
    unsigned long insn, fixup.
}
```

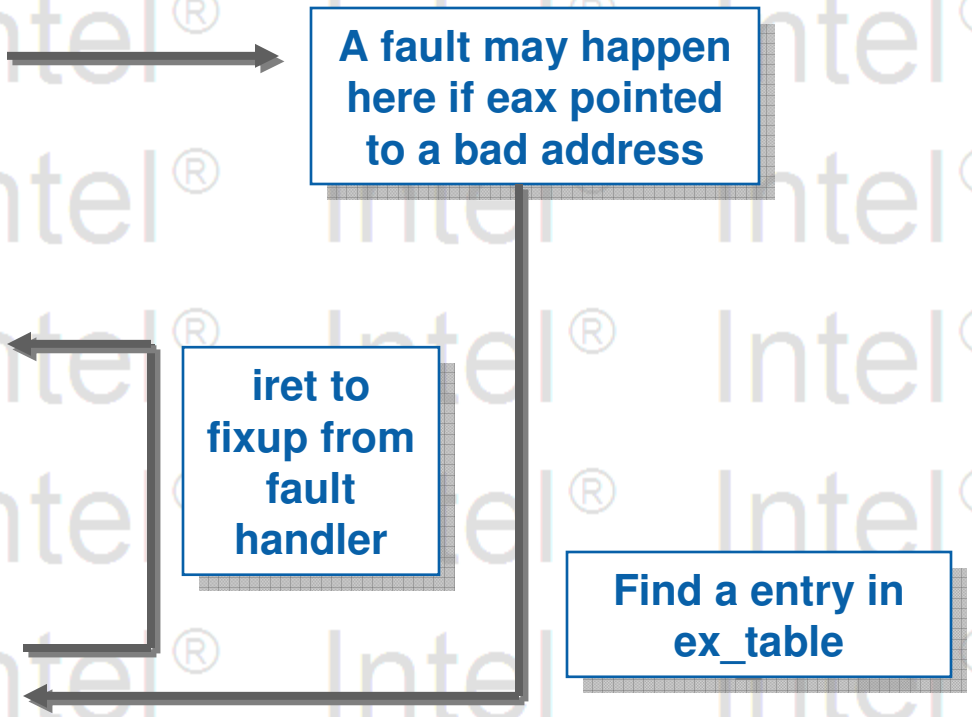
kernel will collect exception table entries at compile time then sort them at startup or module load time.

Virtual Memory

- Exception tables

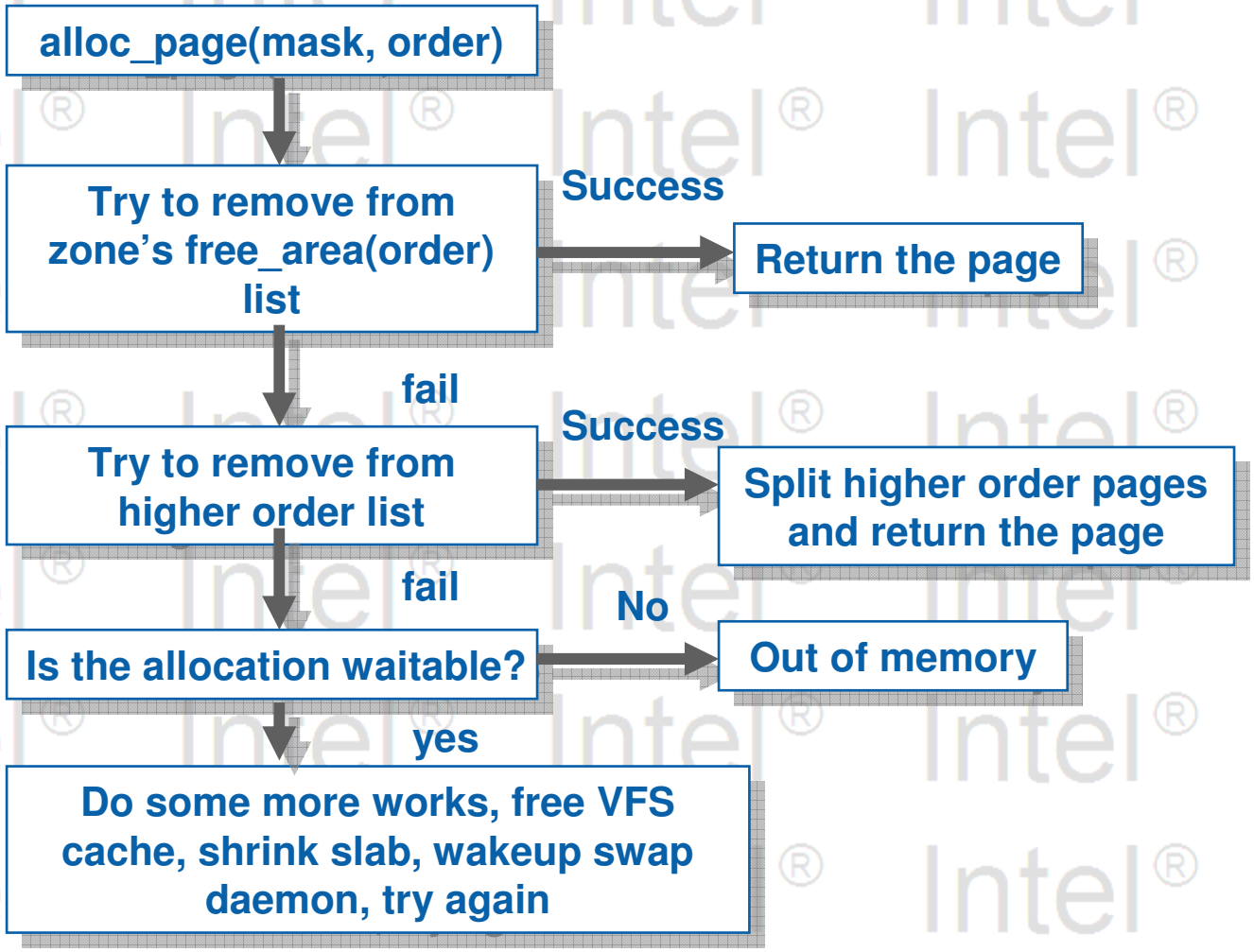
  e.g in __do_clear_user in arch/i386/lib/getuser.S

```
__get_user_1:
    …
1:  movzb1 (%eax), edx
    xorl %eax,%eax
    …
bad_get_user:
    xorl %edx, %edx
    ….
.section _ex_table, “a”
    .long 1b, bad_get_user
```



Virtual Memory

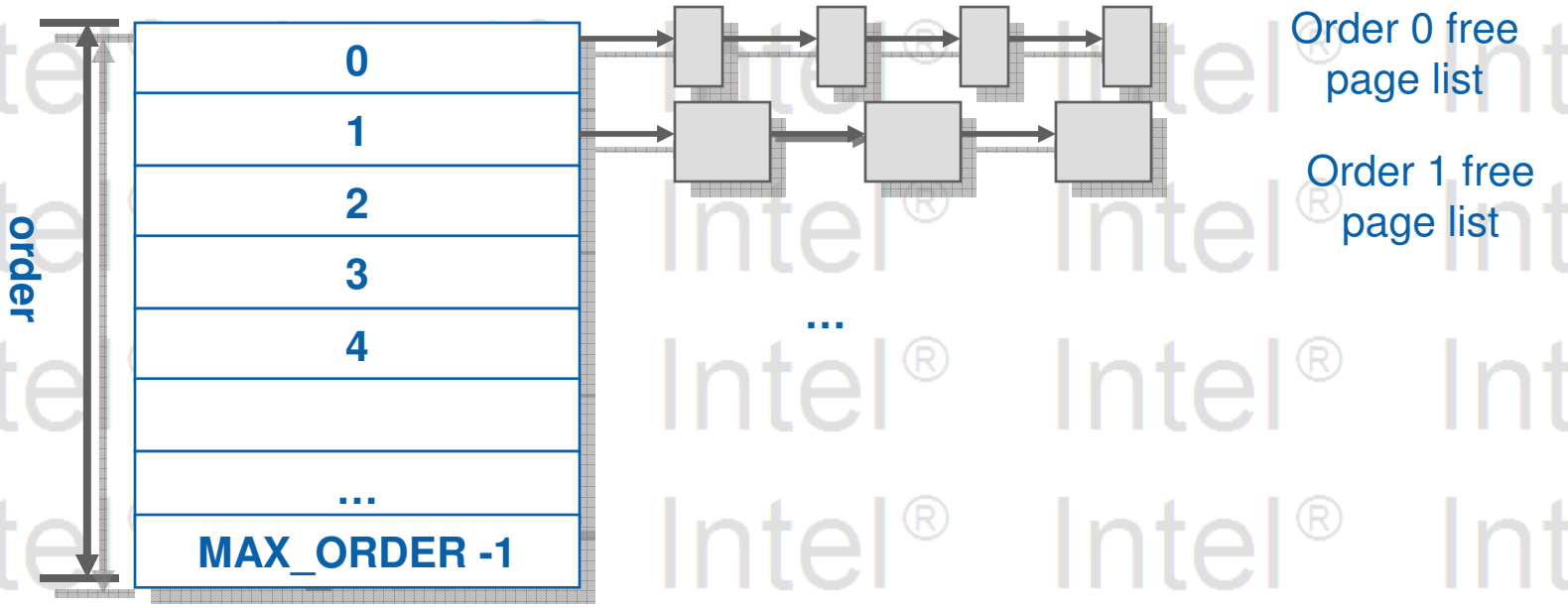
- Simplified page allocation



Virtual Memory

- Linux*'s basic page allocator algorithm, Buddy System

  zone -> free_area



Buddy system

- Allocate from buddy system

  if it is order 0 page try to get page from the per-cpu pcp list.

  if the pcp list empty, kernel will try to grab some pages from per zone free_area list in bulk and put them to per-cpu pcp list.

  if it is order > 0 page, kernel will direct grab pages from per zone free_area.

  if kernel is failed to allocate pages, it will try to do vm balancing if the allocation has the FLAG _GFP_WAIT then retry the allocation.

- Free to buddy system

  if it is an order 0 page, kernel will put page to percpu pcp list, if pcp list is full enough,

  if it is a order > 0 page, kernel will direct put the page back to zone free_area list

  kernel will also try to merge free_area list to higher order list.

SLAB

- Slab allocator

  Slab is an object oriented memory allocator based on page allocator.

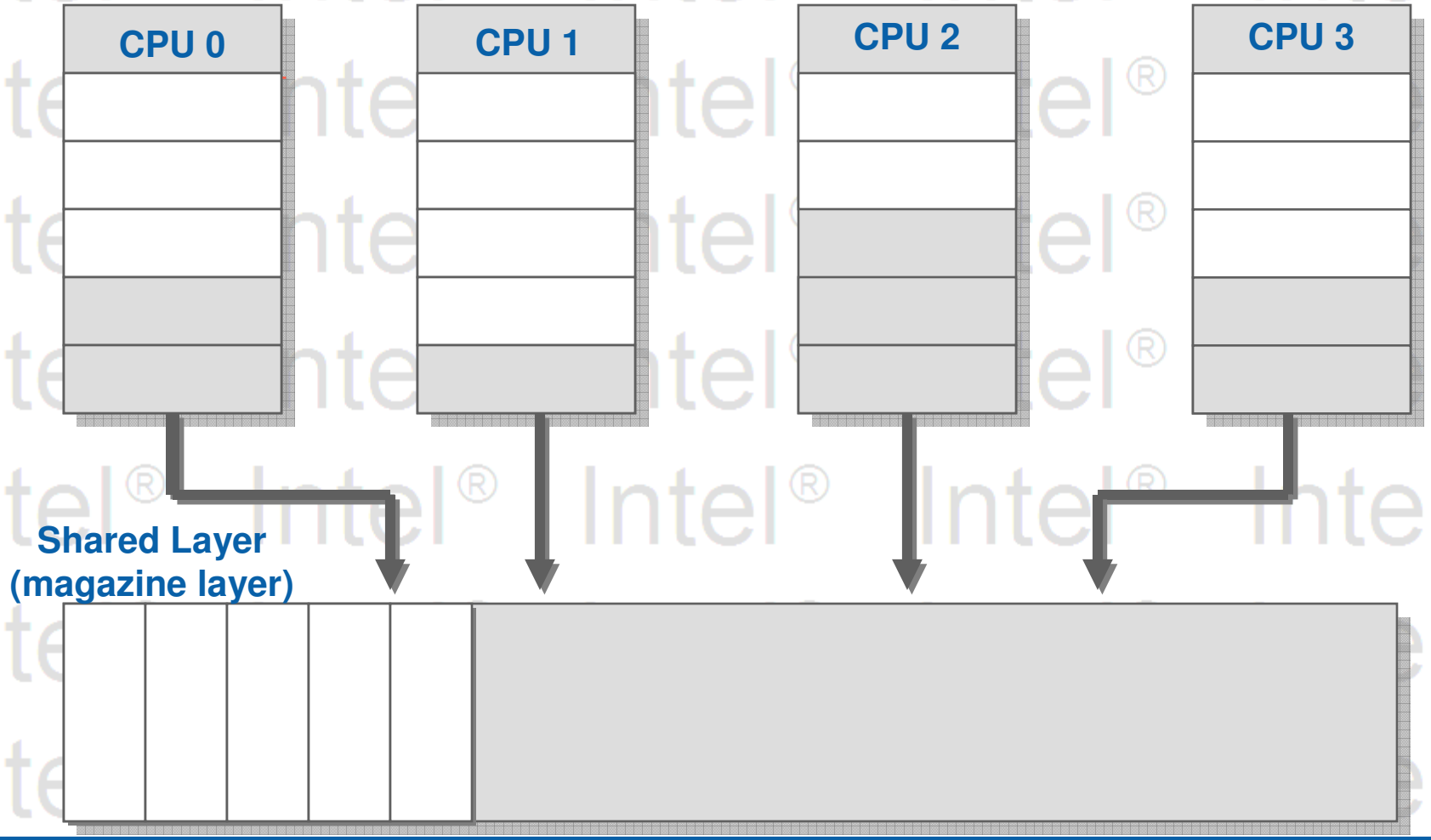
  The idea is kernel is always allocating and freeing some fixed small size of memory blocks, it is better to optimize those cases.

  Slab is better than direct allocate from page allocator in

  1. Slab is lightweight in most hot path.
  2. Slab is cache friendly
  3. Slab is lock friendly

SLAB

SLAB



SLAB

- Slab allocator

  slab colour and alignment
- Kmalloc and Kfree are simple wrapper to slab

File mapping

- File mapping

  each opened file pointer has a pointer pointed to an address space. This address space belongs to the inode of the file. There is a page cache organized with radix tree.

  radix tree is a data structure that can easily insert, delete or find a page via an index. Here the index is the file pointer offset >> PAGE_CACHE_OFFSET.

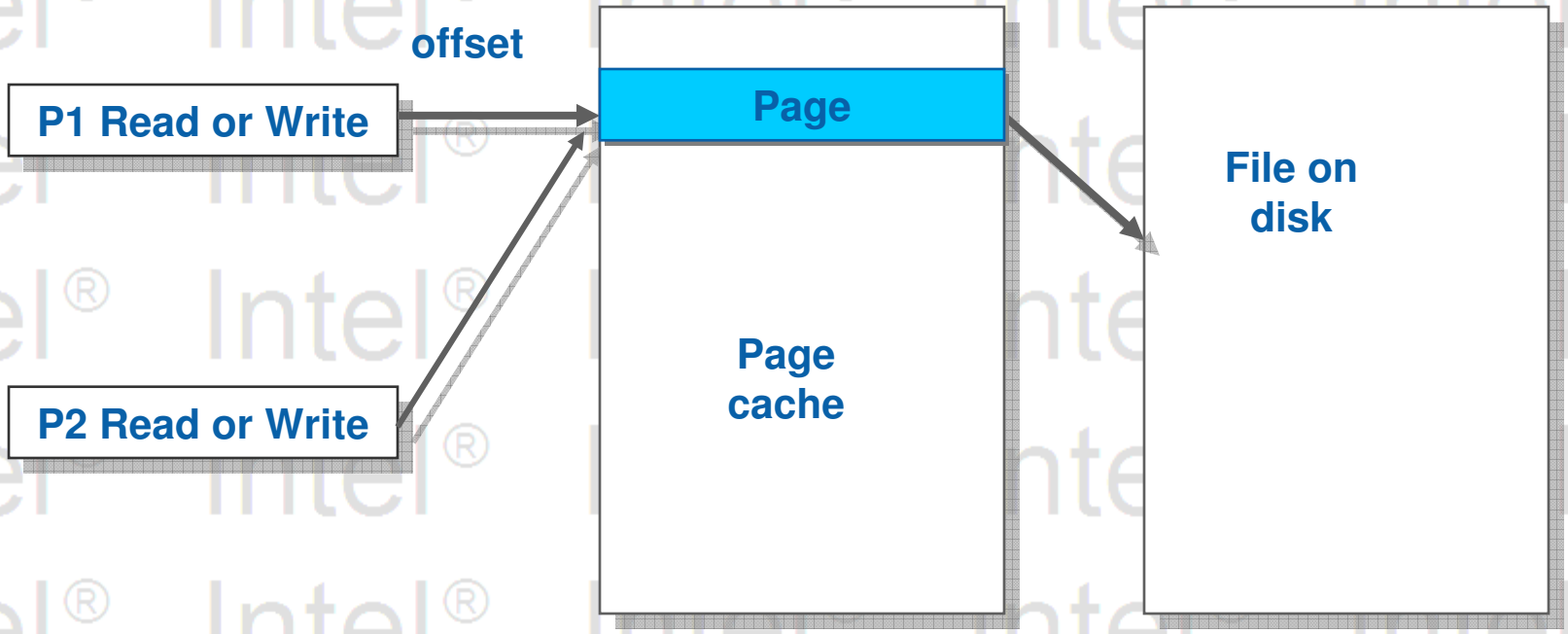
  When a page is begin write to a file, if the file is not opened with O_DIRECT flag, kernel will first lookup the page in mapping->page_trees, if hit, modify the hit page in page_cache mark the page as dirty, otherwise add an entry to the page_cache.

  When a page is being read from a file, if the file is not opened with O_DIRECT flag. kernel will first try to find the page in page cache, if it fails, kernel will issue real I/O read then add the new read page to page_cache.

File mapping

- File mapping

  1 page may belongs to many address spaces, this information is in page->mapping.



Virtual Memory

- VM balancing

There a 2 kind of shrink when system is run out of memory.

the first is shrink_slab which will go through the slabs shrinker list. Each of the shrinker will try to free all the slab memory he owns. Usually those are caches, like dentry cache and inode cache.

the second is shrink zone

each zone has 2 list

active list and inactive list, the 2 lists are organized with LRU manner.

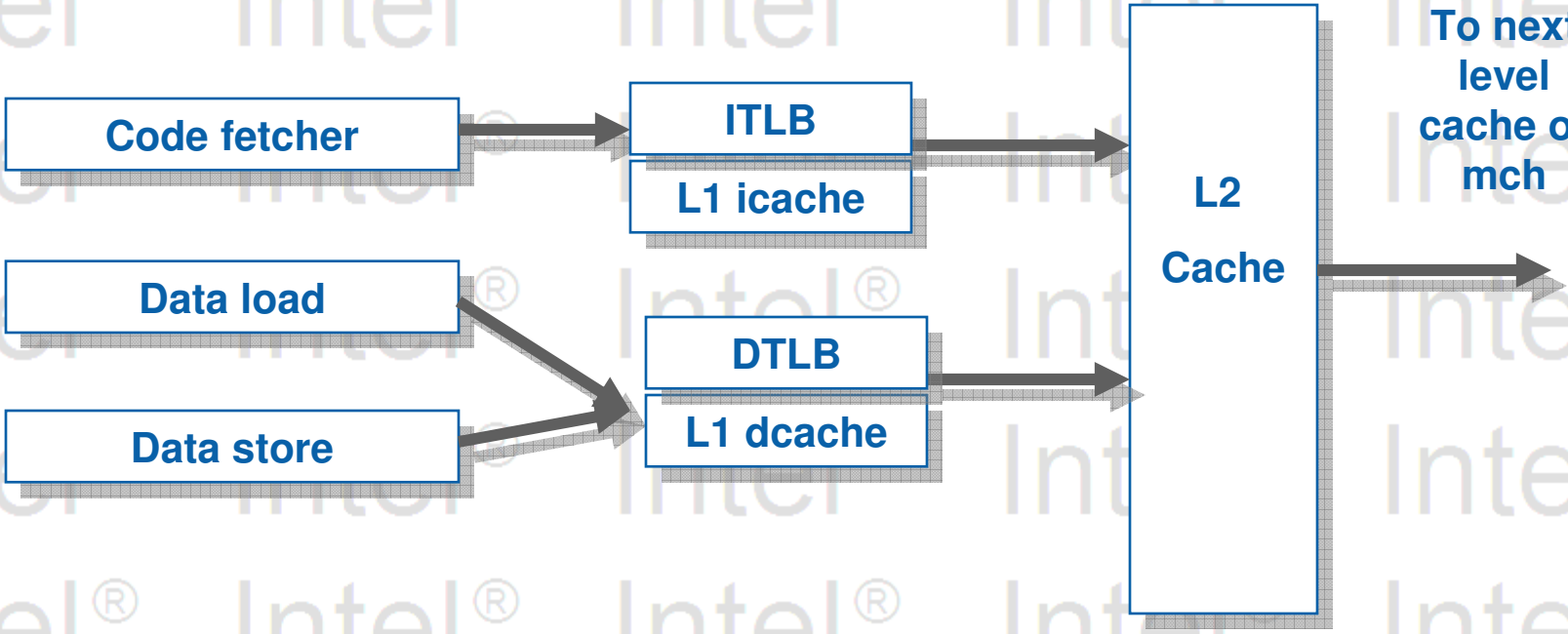
Any fresh allocated pages either via demand paging or COW will be added to the active list.

shrink zone will first scan a certain amount of those pages are not mapped by any processes, drop them to inactive list.

then scan the inactive list, put some of them to swap to back to activelist.

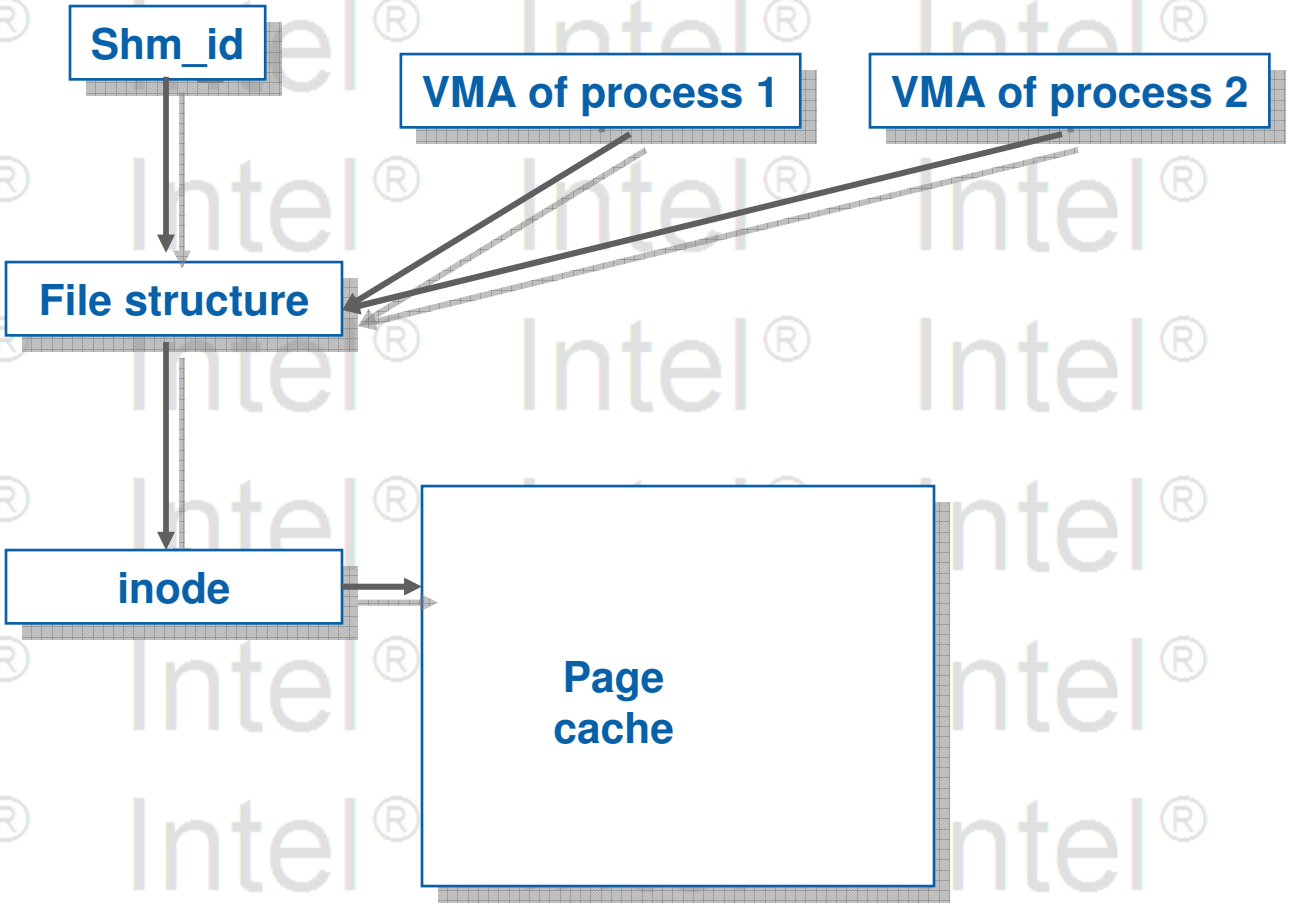
swapper will write back those pages which as dirty flag to swap devices.

Virtual Memory



IPC share memory

- Shared memory
- Shmat shmdt, shmget



Summary

- Kernel/user space memory copy.
- Memory allocators
- Page cache and file mapping
- Share memory
- Reference code lib/extable.c arch/i386/mm/fault.c, mm/vmscan.c, mm/slab.c mm/page_alloc.c, mm/filemap.c


intel®
Leap ahead™