

Memory Management

Advanced Operating Systems and Virtualization

Alessandro Pellegrini

A.Y. 2019/2020



SAPIENZA
UNIVERSITÀ DI ROMA

Memory Management

- During the boot, the Kernel relies on a temporary memory manager
 - It's compact and not very efficient
 - The rationale is that there are not many memory requests during the boot
- At steady state this is no longer the case
 - Allocations/deallocations are frequent
 - Memory must be used wisely, accounting for hardware performance
- We must also discover how much physical memory is available, and how it is organized



NUMA Nodes Organization

- A node is organized in a `struct pglist_data` (even in the case of UMA) typedef'd to `pg_data_t`
- Every node in the system is kept on a NULL-terminated list called `pgdat_list`
- Each node is linked to the next with the field `pg_data_t→node_next`
 - In UMA systems, only one static `pg_data_t` structure called `contig_page_data` is used (defined at `mm/numa.c`)



NUMA Nodes Organization

- From Linux 2.6.16 to 2.6.17 much of the codebase of this portion of the kernel has been rewritten
- Introduction of macros to iterate over node data (most in `include/linux/mmzone.h`) such as:
 - `for_each_online_pgdat()`
 - `first_online_pgdat()`
 - `next_online_pgdat(pgdat)`
- Global `pgdat_list` has since then been removed
- Macros rely on the global `struct pglist_data *node_data[]`;



pg_data_t

- Defined in include/linux/mmzone.h

```
typedef struct pglist_data {  
    zone_t node_zones[MAX_NR_ZONES];  
    zonelist_t node_zonelists[GFP_ZONEMASK+1];  
    int nr_zones;  
    struct page *node_mem_map;  
    unsigned long *valid_addr_bitmap;  
    struct bootmem_data *bdata;  
    unsigned long node_start_paddr;  
    unsigned long node_start_mapnr;  
    unsigned long node_size;  
    int node_id;  
    struct pglist_data *node_next;  
} pg_data_t;
```



Zones

- Nodes are divided into zones:

```
#define ZONE_DMA      0
#define ZONE_NORMAL   1
#define ZONE_HIGHMEM  2
#define MAX_NR_ZONES  3
```

- They target specific physical memory areas:

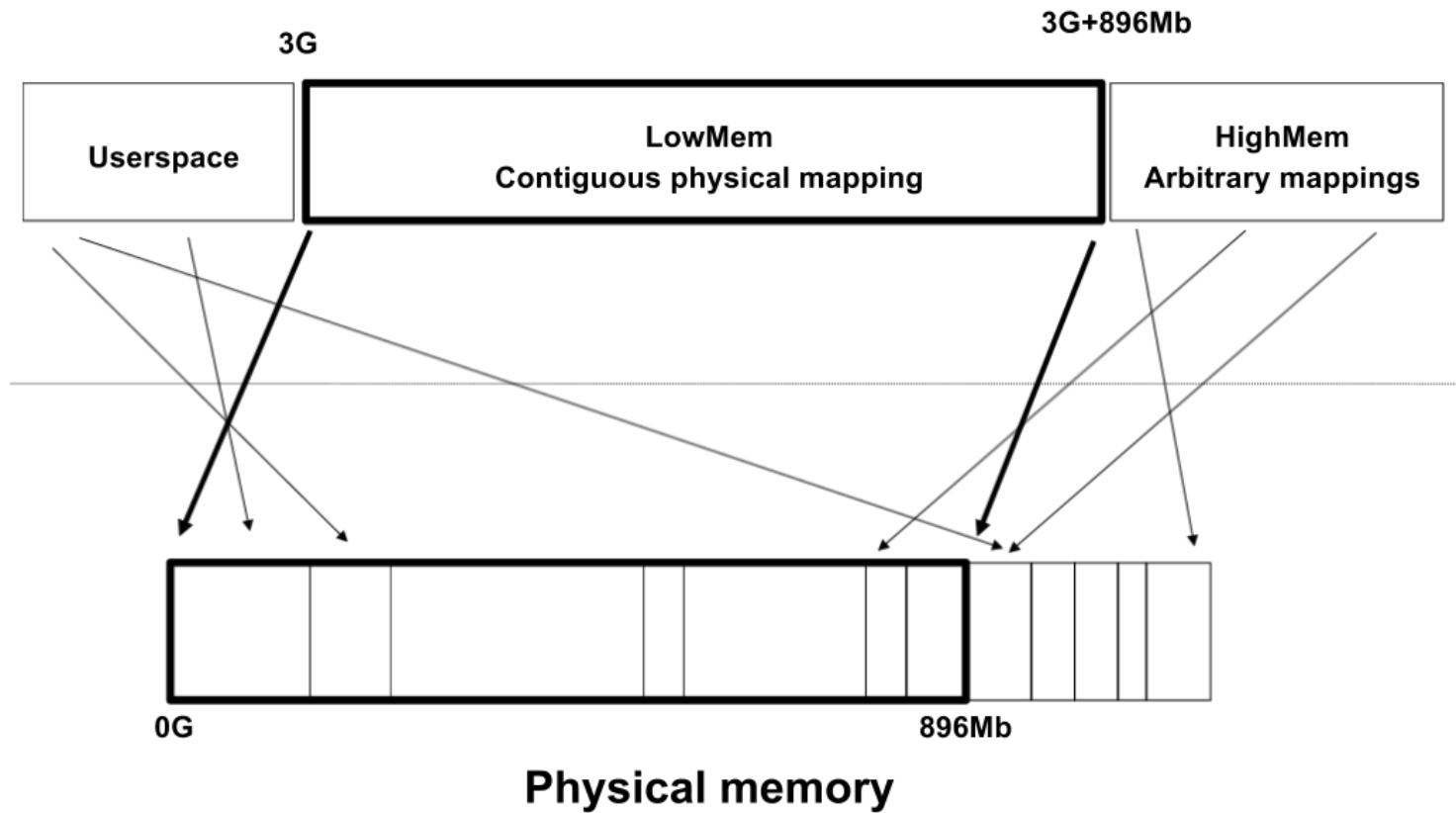
- ZONE_DMA: < 16 MB
- ZONE_NORMAL: 16-896 MB
- ZONE_HIGHMEM: > 896 MB

← Limited in size and high contention. Linux also has the notion of *high memory*



Zones

Virtual memory



Zones Initialization

- Zones are initialized after the kernel page tables have been fully set up by `paging_init()`
- The goal is to determine what parameters to send to:
 - `free_area_init()` for UMA machines
 - `free_area_init_node()` for NUMA machines
- The initialization grounds on PFNs
- max PFN is read from BIOS e820 table



e820 dump in dmesg

```
[0.000000] e820: BIOS-provided physical RAM map:  
[0.000000] BIOS-e820: [mem 0x0000000000000000-0x000000000009fbff] usable  
[0.000000] BIOS-e820: [mem 0x00000000000f0000-0x0000000000ffffff] reserved  
[0.000000] BIOS-e820: [mem 0x0000000000100000-0x0000000007dc08bff] usable  
[0.000000] BIOS-e820: [mem 0x0000000007dc08c00-0x0000000007dc5cbff] ACPI NVS  
[0.000000] BIOS-e820: [mem 0x0000000007dc5cc00-0x0000000007dc5ebff] ACPI data  
[0.000000] BIOS-e820: [mem 0x0000000007dc5ec00-0x0000000007fffffff] reserved  
[0.000000] BIOS-e820: [mem 0x00000000e0000000-0x00000000effffffff] reserved  
[0.000000] BIOS-e820: [mem 0x00000000fec00000-0x00000000fed003ff] reserved  
[0.000000] BIOS-e820: [mem 0x00000000fed20000-0x00000000fed9ffff] reserved  
[0.000000] BIOS-e820: [mem 0x00000000fee00000-0x00000000feeffffff] reserved  
[0.000000] BIOS-e820: [mem 0x00000000ffb00000-0x00000000ffffffff] reserved
```



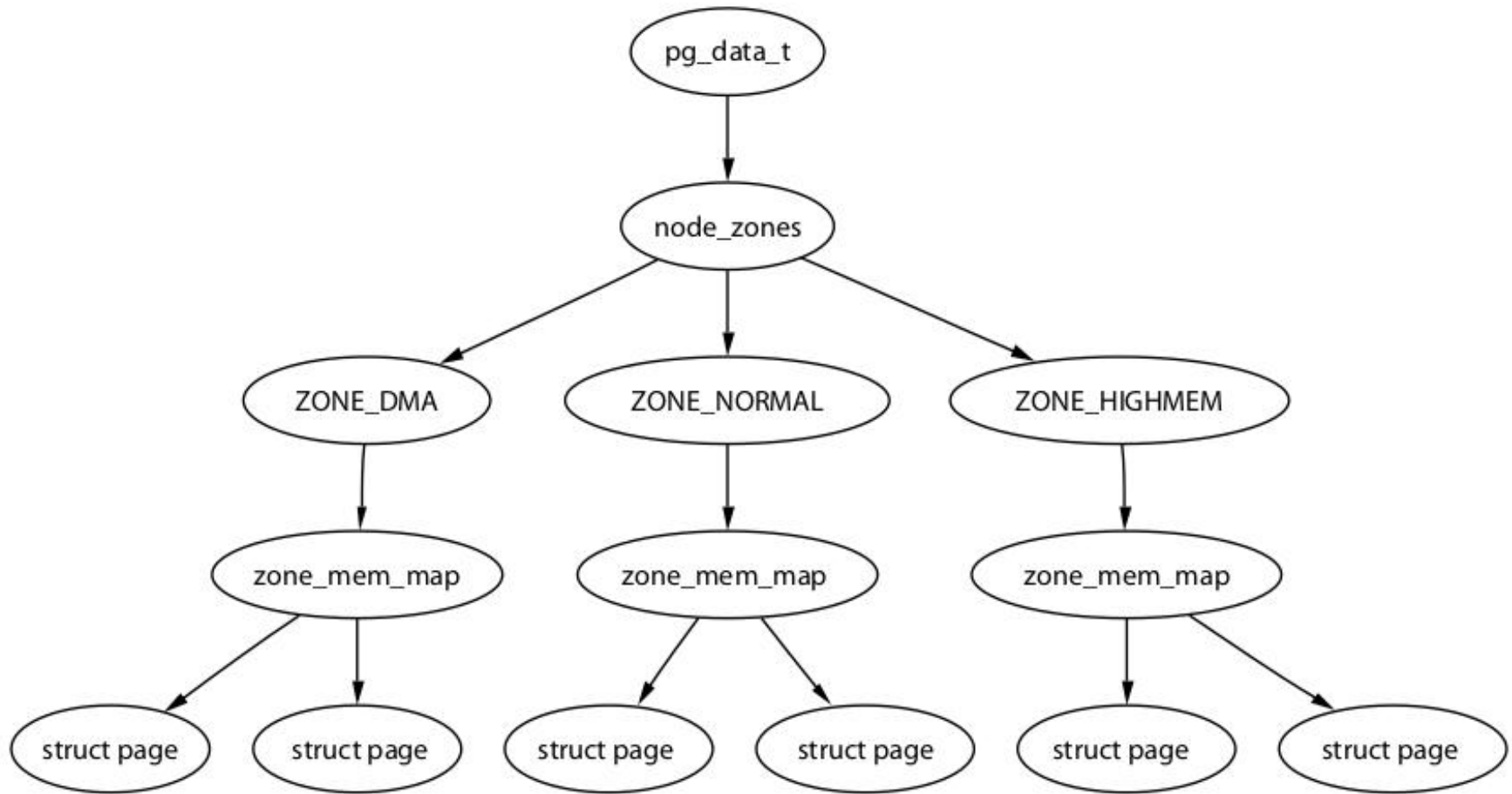
zone_t

```
typedef struct zone_struct {
    spinlock_t          lock;
    unsigned            long    free_pages;
    zone_watermarks_t    watermarks[MAX_NR_ZONES];
    unsigned long        need_balance;
    unsigned long        nr_active_pages, nr_inactive_pages;
    unsigned long        nr_cache_pages;
    free_area_t          free_area[MAX_ORDER];
    wait_queue_head_t    * wait_table;
    unsigned long        wait_table_size;
    unsigned long        wait_table_shift;
    struct pglist_data    *zone_pgdat;
    struct page          *zone_mem_map;
    unsigned long        zone_start_paddr;
    unsigned long        zone_start_mapnr;
    char                 *name;
    unsigned long        size;
    unsigned long        realsize;
} zone_t;
```

Currently 11



Nodes, Zones and Pages Relations



Core Map

- It is an array of `mem_map_t` structures defined in `include/linux/mm.h` and kept in `ZONE_NORMAL`

```
typedef struct page {  
    struct list_head list;                /* ->mapping has some page lists. */  
    struct address_space *mapping;         /* The inode (or ...) we belong to. */  
    unsigned long index;                  /* Our offset within mapping. */  
    struct page *next_hash;               /* Next page sharing our hash bucket in  
                                           the pagecache hash table. */  
  
    atomic_t count;                      /* Usage count, see below. */  
    unsigned long flags;                 /* atomic flags, some possibly  
                                           updated asynchronously */  
  
    struct list_head lru;                  /* Pageout list, eg. active_list;  
                                           protected by pagemap_lru_lock !! */  
  
    struct page **pprev_hash; /* Complement to *next_hash. */  
    struct buffer_head * buffers;         /* Buffer maps us to a disk block. */  
  
    #if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)  
    void *virtual;                        /* Kernel virtual address (NULL if  
                                           not kmapped, ie. highmem) */  
  
    #endif /* CONFIG_HIGMEM || WANT_PAGE_VIRTUAL */  
} mem_map_t;
```



Core Map Members

- Struct members are used to keep track of the interactions between MM and other kernel sub-systems
- `struct list_head list`: used to organize the frames into free lists
- `atomic_t count`: counts the virtual references mapped onto the frame
- `unsigned long flags`: status bits for the frame
 - `#define PG_locked 0`
 - `#define PG_referenced 2`
 - `#define PG_uptodate 3`
 - `#define PG_dirty 4`
 - `#define PG_lru 6`
 - `#define PG_reserved 14`



How to manage flags

Bit Name	Set	Test	Clear
PG_active	SetPageActive()	PageActive()	ClearPageActive()
PG_arch_1	None	None	None
PG_checked	SetPageChecked()	PageChecked()	None
PG_dirty	SetPageDirty()	PageDirty()	ClearPageDirty()
PG_error	SetPageError()	PageError()	ClearPageError()
PG_highmem	None	PageHighMem()	None
PG_laundry	SetPageLaundry()	PageLaundry()	ClearPageLaundry()
PG_locked	LockPage()	PageLocked()	UnlockPage()
PG_lru	TestSetPageLRU()	PageLRU()	TestClearPageLRU()
PG_referenced	SetPageReferenced()	PageReferenced()	ClearPageReferenced()
PG_reserved	SetPageReserved()	PageReserved()	ClearPageReserved()
PG_skip	None	None	None
PG_slab	PageSetSlab()	PageSlab()	PageClearSlab()
PG_unused	None	None	None
PG_uptodate	SetPageUptodate()	PageUptodate()	ClearPageUptodate()



Core Map on UMA

- Initially we only have the core map pointer
- This is `mem_map` and is declared in `mm/memory.c`
- Pointer initialization and corresponding memory allocation occur within `free_area_init()`
- After initializing, each entry will keep the value 0 within the `count` field and the value 1 into the `PG_reserved` flag within the `flags` field
- Hence no virtual reference exists for that frame and the frame is reserved
- Frame un-reserving will take place later via the function `mem_init()` in `arch/i386/mm/init.c` (by resetting the bit `PG_reserved`)



Core Map on NUMA

- There is not a global `mem_map` array
- Every node keeps its own map in its own memory
- This map is referenced by `pg_data_t→node_mem_map`
- The rest of the organization of the map does not change



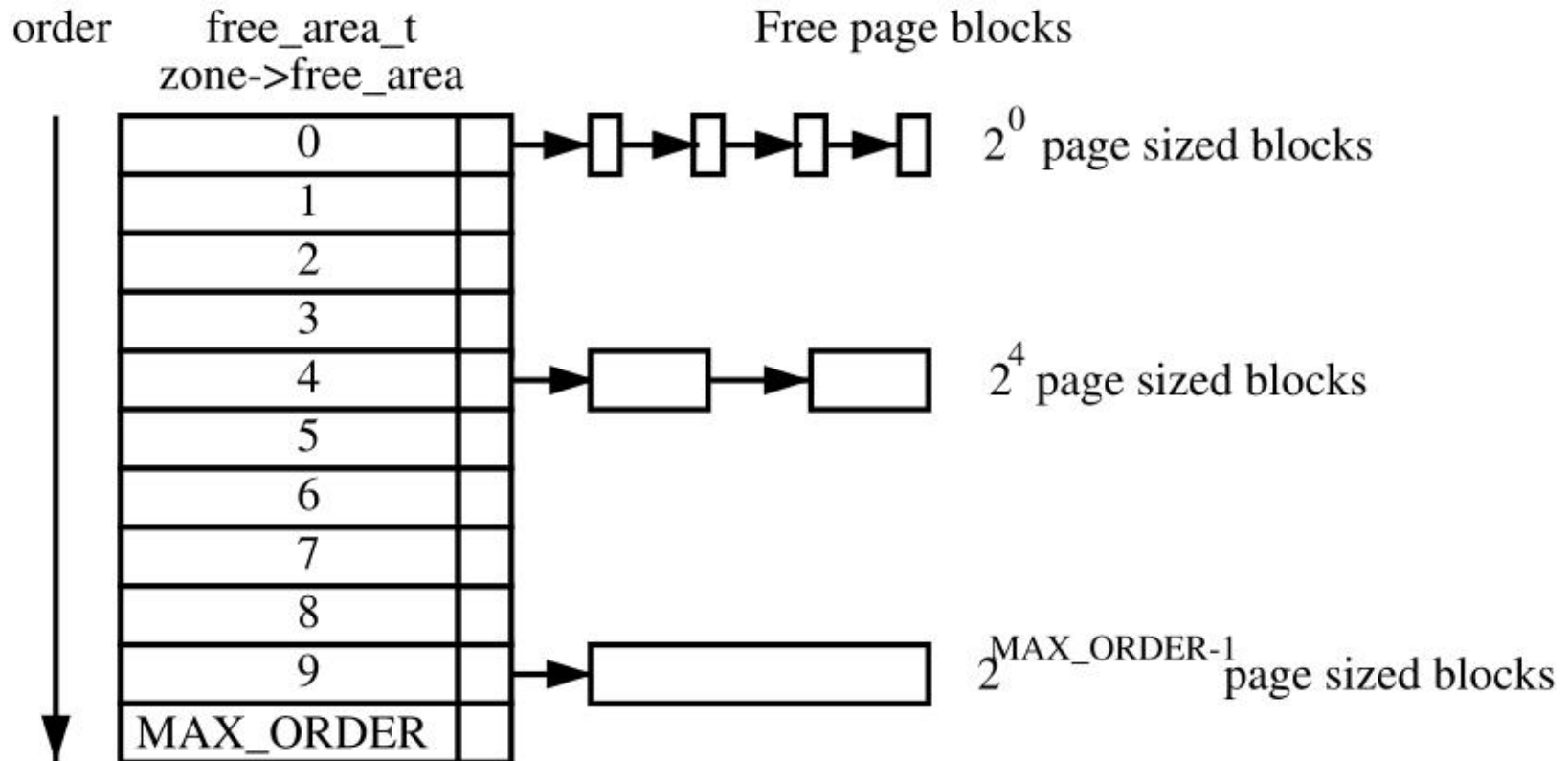
Buddy System: Frame Allocator

- By Knowlton (1965) and Knuth (1968)
- It has been experimentally shown to be quite fast
- Based on two main data structures:

```
typedef struct free_area_struct {  
    struct list_head list;  
    unsigned int *map;  
} free_area_t  
  
struct list_head {  
    struct list_head *next, *prev;  
}
```



free_area_t organization



Bitmap $*_{map}$ semantic

- Linux saves memory by using one bit for a pair of buddies
- It's a "fragmentation" bit
- Each time a buddy is allocated or free'd, the bit representing the pair is toggled
 - 0: if the pages are both free or allocated
 - 1: only one buddy is in use

Index in the global
mem_map array



```
#define MARK_USED(index, order, area) \  
__change_bit((index) >> (1+(order)), (area)->map)
```



High Memory

- When the size of physical memory approaches/ exceeds the maximum size of virtual memory, it is impossible for the kernel to keep all of the available physical memory mapped
- “Highmem” is the memory not covered by a permanent mapping
- The Kernel has an API to allow “temporary mappings”
- This is where userspace memory comes from



High Memory

- `vmap()` : used to make a long-duration mapping of multiple physical pages
- `kmap()` : it permits a short-duration mapping of a single page.
 - It needs global synchronization, but is amortized somewhat.
- `kmap_atomic()` : This permits a very short duration mapping of a single page.
 - It is restricted to the CPU that issued it
 - the issuing task is required to stay on that CPU until it has finished
- In general: nowadays, it *really* makes sense to use 64-bit systems!



High Memory Deallocation

- Kernel maintains an array of counters:

```
static int pkmap_count[ LAST_PKMAP ];
```

- One counter for each 'high memory' page
- Counter values are 0, 1, or more than 1:
 - =0: page is not mapped
 - =1: page not mapped now, but used to be
 - =n > 1: page was mapped (n-1) times



kunmap ()

- `kunmap (page)` decrements the associated reference counter
- When the counter is 1, mapping is not needed anymore
- But CPU still has “cached” that mapping
- So the mapping must be “invalidated”
- With multiple CPUs, all of them must do it
 - `__flush_tlb_all()`



Reclaiming Boot Memory

- The finalization of memory management init is done via `mem_init()` which destroys the bootmem allocator
- This function will release the frames, by resetting the `PG_RESERVED` bit
- For each free'd frame, the function `__free_page()` is invoked
 - This gives all the pages in `ZONE_NORMAL` to the buddy allocator
- At this point the reference `count` within the corresponding entry gets set to 1 since the kernel maps that frame anyway within its page table



Finalizing Memory Initialization

```
static unsigned long __init
free_all_bootmem_core(pg_data_t *pgdat) {
    .....
    // Loop through all pages in the current node
    for (i = 0; i < idx; i++, page++) {
        if (!test_bit(i, bdata->node_bootmem_map)) {
            count++;
            ClearPageReserved(page);
            // Fake the buddy into thinking it's an
            // actual free
            set_page_count(page, 1);
            __free_page(page);
        }
    }
    total += count;
    .....
    return total;
}
```



Allocation Contexts

- Process context: allocation due to a system call
 - If it cannot be served: wait along the current execution trace
 - Priority-based approach
- Interrupt: allocation due to an interrupt handler
 - If it cannot be served: no actual waiting time
 - Priority independent schemes
- This approach is general to most Kernel subsystems



Basic Kernel Internal MM API

- At steady state, the MM subsystem exposes API to other kernel subsystems
- Prototypes in `#include <linux/malloc.h>`
- Basic API: page allocation
 - `unsigned long get_zeroed_page(int flags):` take a frame from the free list, zero the content and return its virtual address
 - `unsigned long __get_free_page(int flags):` take a frame from the free list and return its virtual address
 - `unsigned long __get_free_pages(int flags, unsigned long order):` take a block of contiguous frames of given order from the free list



Basic Kernel Internal MM API

- Basic API: page allocation
 - `void free_page(unsigned long addr):`
put a frame back into the free list
 - `void free_pages(unsigned long addr,
unsigned long order):` put a block of frames
of given order back into the free list
- Warning: passing a wrong `addr` or `order`
might corrupt the Kernel!



Basic Kernel Internal MM API

- `flags`: used to specify the allocation context
 - `GFP_ATOMIC`: interrupt context. The call cannot lead to sleep
 - `GFP_USER`: Used to allocate memory for userspace-related activities. The call can lead to sleep
 - `GFP_BUFFER`: Used to allocate a buffer. The call can lead to sleep
 - `GFP_KERNEL`: Used to allocate Kernel memory. The call can lead to sleep



NUMA Allocation

- On NUMA systems, we have multiple nodes
- UMA systems eventually invoke NUMA API, but the system is configured to have a single node
- Core memory allocation API:
 - `struct page *alloc_pages_node(int nid, unsigned int flags, unsigned int order);`
 - `__get_free_pages()` **calls** `alloc_pages_node()` specifying a *NUMA policy*



NUMA Policies

- NUMA policies determine what NUMA node is involved in a memory operation
- Since Kernel 2.6.18, userspace can tell the Kernel what policy to use:

```
#include <numaif.h>
int set_mempolicy(int mode, unsigned long
*nodemask, unsigned long maxnode);
```

- mode **can be:** MPOL_DEFAULT, MPOL_BIND, MPOL_INTERLEAVE **or** MPOL_PREFERRED



NUMA Policies

```
#include <numaif.h>
int mbind(void *addr, unsigned long len,
int mode, unsigned long *nodemask,
unsigned long maxnode, unsigned flags);
```

Sets the NUMA memory policy, which consists of a policy mode and zero or more nodes, for the memory range starting with *addr* and continuing for *len* bytes. The memory policy defines from which node memory is allocated.



Moving Pages Around

```
#include <numaif.h>

long move_pages(int pid, unsigned long
count, void **pages, const int *nodes,
int *status, int flags);
```

Moves the specified *pages* of the process *pid* to the memory nodes specified by *nodes*. The result of the move is reflected in *status*. The *flags* indicate constraints on the pages to be moved.



Frequent Allocations/Deallocations

- Consider fixed-size data structures which are frequently allocated/released
- The buddy system here does not scale
 - This is a classical case of frequent logical contention
 - The Buddy System on each NUMA node is protected by a spinlock
 - The internal fragmentation might rise too much



Classical Examples

- Allocation/release of page tables, at any level, is very frequent
- We want to perform these operations quickly
- For paging we have:
 - `pgd_alloc()`, `pmd_alloc()` and `pte_alloc()`
 - `pgd_free()`, `pmd_free()` and `pte_free()`
- They rely on one of Kernel-level *fast allocators*



Fast Allocation

- There are several fast allocators in the Kernel
- For paging, there are the *quicklists*
- For other buffers, there is the *slab allocator*
- There are three implementations of the slab allocator in Linux:
 - the SLAB: Implemented around 1994
 - the SLUB: The Unqueued Slab Allocator, default since Kernel 2.6.23
 - the SLOB: Simple List of Blocks. If the SLAB is disabled at compile time, Linux reverts to this



Quicklist

- Defined in `include/linux/quicklist.h`
- They are implemented as a list of per-core page lists
- There is no need for synchronization
- If allocation fails, they revert to `__get_free_page()`



Quicklist Allocation

```
static inline void *quicklist_alloc(int nr, gfp_t flags, ...) {
    struct quicklist *q;
    void **p = NULL;

    q = &get_cpu_var(quicklist)[nr];
    p = q->page;
    if (likely(p)) {
        q->page = p[0];
        p[0] = NULL;
        q->nr_pages--;
    }
    put_cpu_var(quicklist);
    if (likely(p))
        return p;

    p = (void *)__get_free_page(flags | __GFP_ZERO);
    return p;
}
```



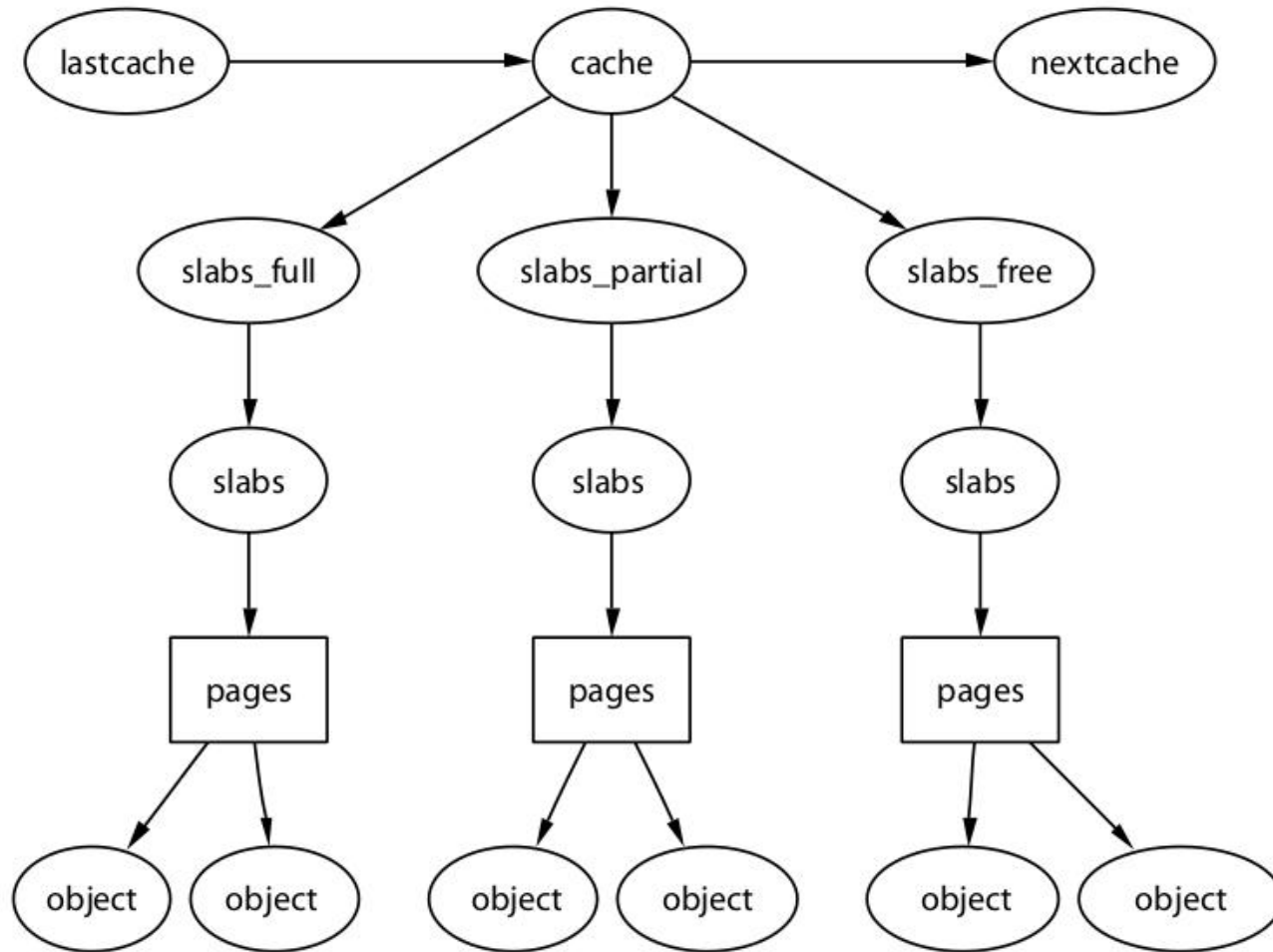
likely/unlikely

- **Defined in** `include/linux/compiler.h`

```
# define likely(x)    __builtin_expect(!!(x), 1)  
# define unlikely(x)  __builtin_expect(!!(x), 0)
```
- `!!` is used to convert any value to 1 or 0
- Up to Pentium 4:
 - `0x2e`: Branch Not Taken
 - `0x3e`: Branch Taken



The SLAB Allocator



SLAB Interfaces

- Prototypes are in `#include <linux/malloc.h>`
- `void *kmalloc(size_t size, int flags):`
allocation of contiguous memory (it returns the virtual address)
- `void kfree(void *obj):` frees memory allocated via `kmalloc()`
- `void *kmalloc_node(size_t size, int flags, int node):` NUMA-aware allocation



Available Caches (up to 3.9.11)

```
struct cache_sizes {
    size_t                cs_size;
    struct kmem_cache     *cs_cache;
#ifdef CONFIG_ZONE_DMA
    struct kmem_cache     *cs_dmacache;
#endif
}
```

```
static cache_sizes_t cache_sizes[] = {
    {32,          NULL,          NULL},
    {64,          NULL,          NULL},
    {128,         NULL,          NULL},
    ...,
    {65536,       NULL,          NULL},
    {131072,      NULL,          NULL},
}
```



Available Caches (since 3.10)

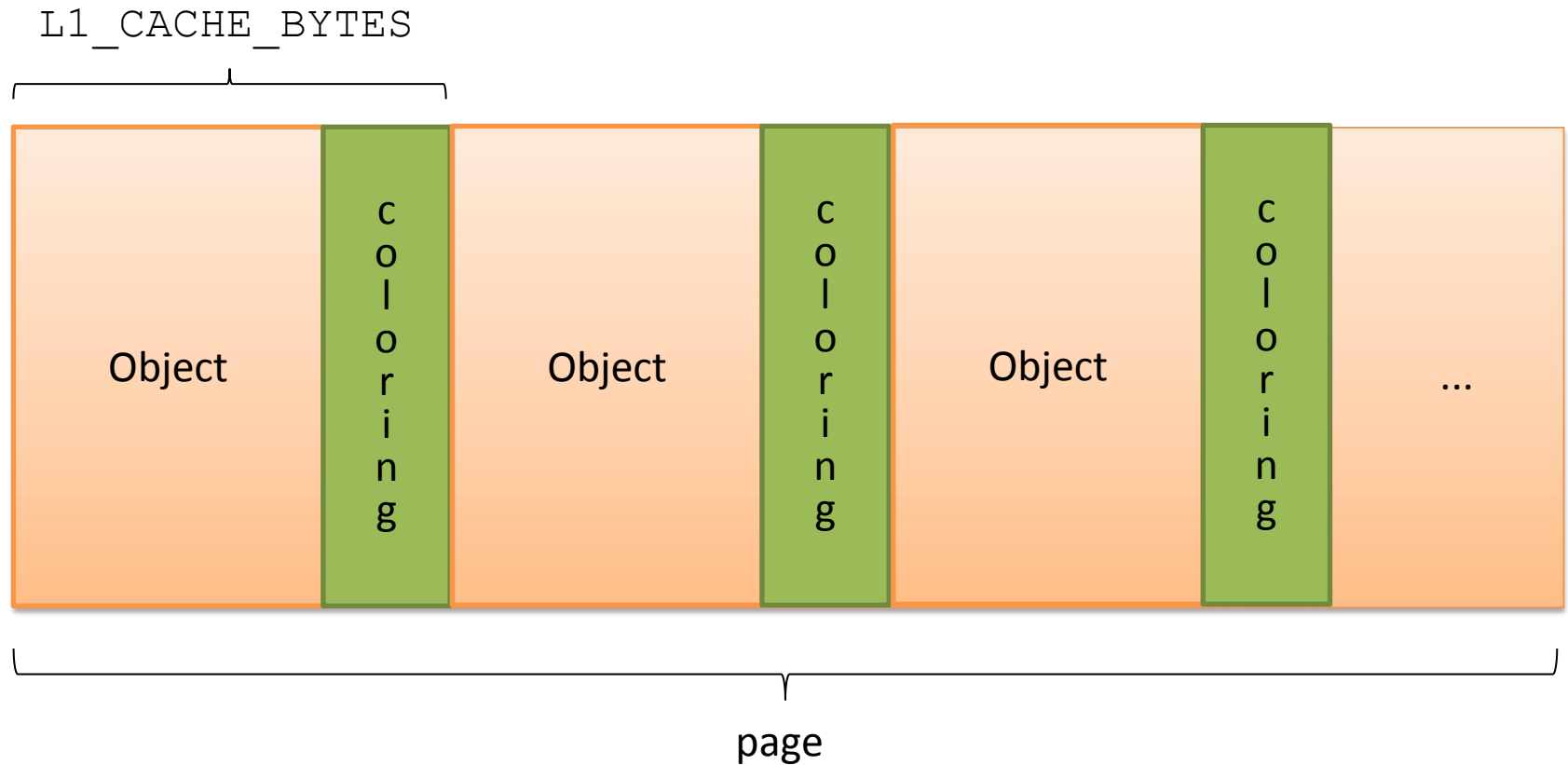
```
struct kmem_cache_node {
    spinlock_t list_lock;

#ifdef CONFIG_SLAB
    struct list_head slabs_partial; /* partial list first, better
                                     asm code */

    struct list_head slabs_full;
    struct list_head slabs_free;
    unsigned long free_objects;
    unsigned int free_limit;
    unsigned int colour_next;      /* Per-node cache coloring */
    struct array_cache *shared;     /* shared per node */
    struct array_cache **alien;     /* on other nodes */
    unsigned long next_reap;        /* updated without locking */
    int free_touched;              /* updated without locking */
#endif
};
```



Slab Coloring



L1 data caches

- Cache lines are small (typically 32/64 bytes)
- `L1_CACHE_BYTES` is the configuration macro in Linux
- Independently of the mapping scheme, close addresses fall in the same line
- Cache-aligned addresses fall in different lines
- We need to cope with *cache performance issues at the level of kernel programming* (typically not of explicit concern for user level programming)



Cache Performance Aspects

- *Common members* access issues
 - Most-used members in a data structure should be placed at its head to maximize cache hits
 - This should happen provided that the slab-allocation (kmalloc()) system gives cache-line aligned addresses for dynamically allocated memory chunks
- *Loosely related fields* should be placed sufficiently distant in the data structure so as to avoid performance penalties due to *false cache sharing*
- The Kernel has also to deal with Aliasing



Cache flush operations

- Cache flushes automation can be partial (similar to TLB)
- Need for explicit cache flush operations
- In some cases, the flush operation uses the physical address of the cached data to support flushing (“strict caching systems”, e.g. HyperSparc)
- Hence, TLB flushes should always be placed after the corresponding data cache flush calls

Flushing Full MM	Flushing Range	Flushing Page
<code>flush_cache_mm()</code> Change all page tables <code>flush_tlb_mm()</code>	<code>flush_cache_range()</code> Change page table range <code>flush_tlb_range()</code>	<code>flush_cache_page()</code> Change single PTE <code>flush_tlb_page()</code>



Cache flush operations

- `void flush_cache_all(void)`
 - Flushes the entire CPU cache system, which makes it the most severe flush operation to use
 - It is used when changes to the kernel page tables, which are global in nature, are to be performed
- `void flush_cache_mm(struct mm_struct *mm)`
 - Flushes all entries related to the address space
 - On completion, no cache lines will be associated with `mm`



Cache flush operations

```
void flush_cache_range(struct mm_struct *mm,  
    unsigned long start, unsigned long end)
```

- This flushes lines related to a range of addresses
- Like its TLB equivalent, it is provided in case the architecture has an efficient way of flushing ranges instead of flushing each individual page

```
void flush_cache_page(struct vm_area_struct  
*vma, unsigned long vmaddr)
```

- Flushes a single-page-sized region
- `vma` is supplied because the `mm_struct` is easily accessible through `vma->vm_mm`
- Additionally, by testing for the `VM_EXEC` flag, the architecture knows if the region is executable for caches that separate the instructions and data caches



User-/Kernel-Level Data Movement

```
unsigned long copy_from_user(void *to, const void *from,  
                             unsigned long n)
```

Copies n bytes from the user address(from) to the kernel address space(to).

```
unsigned long copy_to_user(void *to, const void *from,  
                             unsigned long n)
```

Copies n bytes from the kernel address(from) to the user address space(to).

```
void get_user(void *to, void *from)
```

Copies an integer value from userspace (from) to kernel space (to).

```
void put_user(void *from, void *to)
```

Copies an integer value from kernel space (from) to userspace (to).

```
long strncpy_from_user(char *dst, const char *src, long count)
```

Copies a null terminated string of at most count bytes long from userspace (src) to kernel space (dst)

```
int access_ok(int type, unsigned long addr, unsigned  
              long size)
```

Returns nonzero if the userspace block of memory is valid and zero otherwise



Large-size Allocations

- Typically used when adding large-size data structures to the kernel in a stable way
- This is the case when, e.g., mounting external modules
- The main APIs are:
 - `void *vmalloc(unsigned long size)`
allocates memory of a given size, which can be non-contiguous, and returns the virtual address (the corresponding frames are reserved)
 - `void vfree(void *addr)`
frees the above mentioned memory



Logical/Physical Address Translation

- This is valid only for kernel directly mapped memory (not vmalloc'd memory)
- `virt_to_phys(unsigned int addr)` (in `include/x86/io.h`)
- `phys_to_virt(unsigned int addr)` (in `include/x86/io.h`)

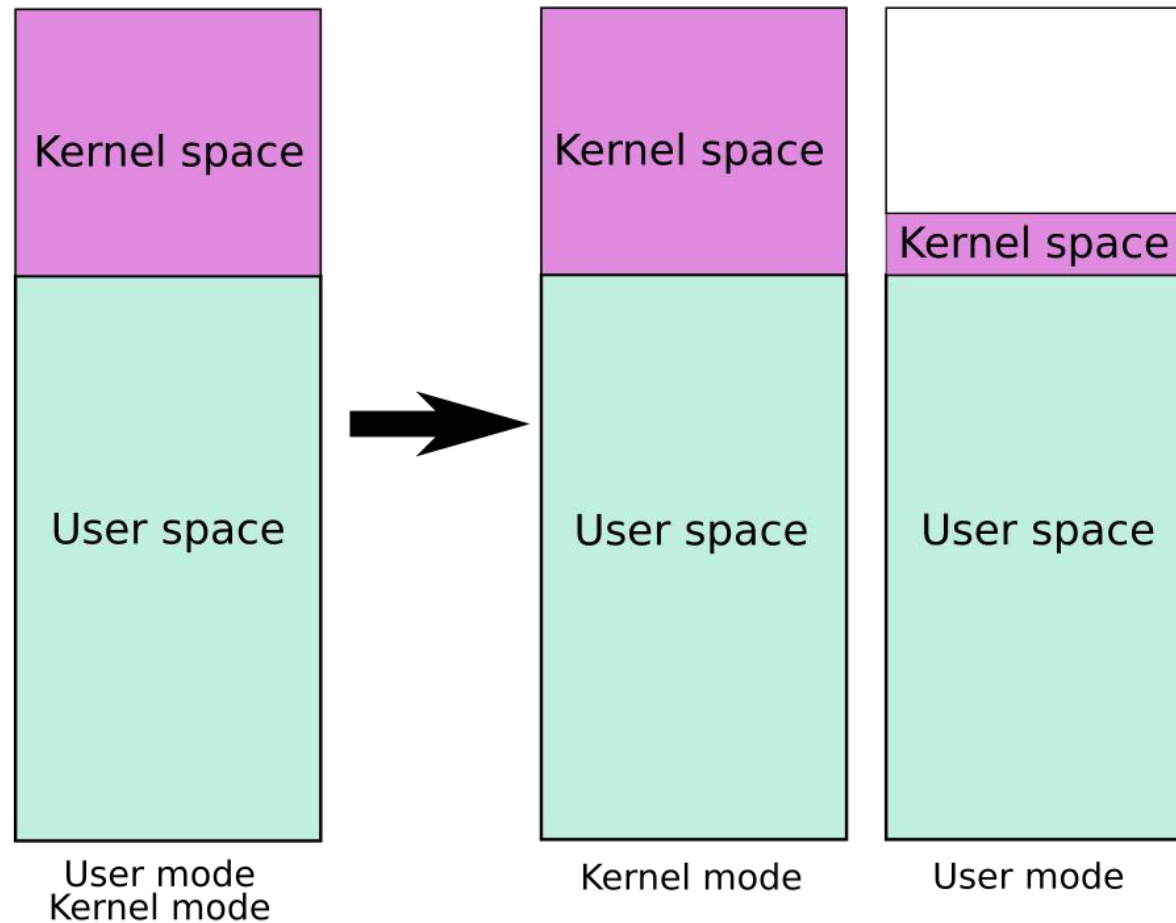


kmalloc() VS vmalloc()

- Allocation size:
 - Bounded for kmalloc (cache aligned)
 - The boundary depends on the architecture and the Linux version. Current implementations handle up to 8KB
 - 64/128 MB for vmalloc
- Physical contiguousness
 - Yes for kmalloc
 - No for vmalloc
- Effects on TLB
 - None for kmalloc
 - Global for vmalloc (transparent to vmalloc users)



Kernel Page Table Isolation



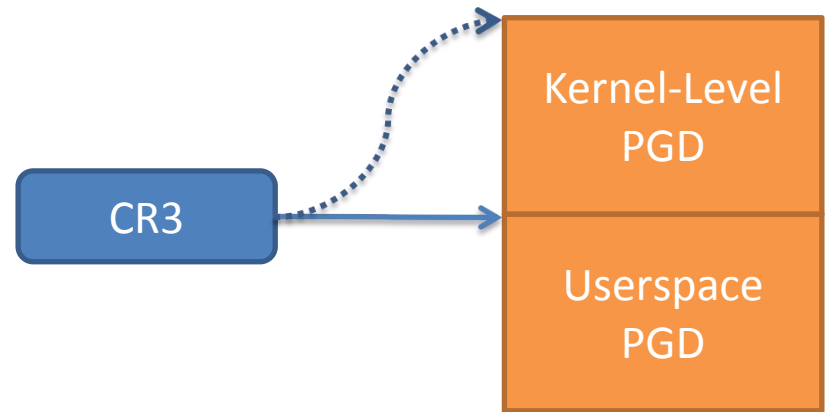
cpu_entry_area

```
struct cpu_entry_area {  
    char gdt[PAGE_SIZE];  
  
    struct entry_stack_page entry_stack_page;  
  
    struct tss_struct tss;  
  
    char exception_stacks[...];  
};  
  
DECLARE_PER_CPU(struct cpu_entry_area *,  
                 cpu_entry_area);
```



Double Page General Directory

- The first level of the page table is composed of a buffer of 8 KBs (two actual pages)
- One page is used to map the kernel-level memory view
- The other one is used to map the userspace memory view



CR3 is updated when transitioning to and from kernel mode



Switch CR3

```
/arch/x86/entry/entry_64.S:
```

```
SYM_CODE_START(entry_SYSCALL_64)
```

```
...
```

```
SWITCH_TO_KERNEL_CR3 scratch_reg=%rsp
```

```
...
```

```
SWITCH_TO_USER_CR3_STACK scratch_reg=%rdi
```

```
/arch/x86/entry/calling.h:
```

```
.macro SWITCH_TO_KERNEL_CR3 scratch_reg:req
```

```
    mov    %cr3, \scratch_reg
```

```
    andq    $(~PTI_USER_PGTABLE_AND_PCID_MASK), \scratch_reg
```

```
    mov    \scratch_reg, %cr3
```

```
.endm
```



Memory View Consistency

- When a minor fault occurs, the process transitions to kernel mode
- Any update in the page table is therefore reflected only in the kernel-view page table
- The userspace page table must be explicitly realigned by the fault handler
- This same behavior occurs when a clone of the page table is created (this aspect is related to the implementation of the `fork()`)



Memory View Consistency (5.5)

```
static ninline int vmalloc_fault(unsigned long address)
{
    pgd_t *pgd, *pgd_k;
    p4d_t *p4d, *p4d_k;
    pud_t *pud;
    pmd_t *pmd;
    pte_t *pte;

    pgd = (pgd_t *) __va(read_cr3_pa()) + pgd_index(address);

    if (pgd_none(*pgd)) {
        set_pgd(pgd, *pgd_k);
    }

    ...
}
```



Memory View Consistency (5.5)

```
/arch/x86/include/asm/pgtable_64.h:
```

```
static inline void native_set_pgd(pgd_t *pgdp, pgd_t pgd)
{
    WRITE_ONCE(*pgdp, pti_set_user_pgtbl(pgdp, pgd));
}
```

```
/arch/x86/mm/pti.c:
```

```
pgd_t __pti_set_user_pgtbl(pgd_t *pgdp, pgd_t pgd)
{
    if (!pgdp_maps_userspace(pgdp))
        return pgd;

    kernel_to_user_pgdp(pgdp)->pgd = pgd.pgd;

    if ((pgd.pgd & (_PAGE_USER|_PAGE_PRESENT)) == (_PAGE_USER|_PAGE_PRESENT) &&
        (__supported_pte_mask & _PAGE_NX))
        pgd.pgd |= _PAGE_NX;

    return pgd;
}
```

