Memory Management - (25-03-2020)

After its initialization the kernel has to setup one fundamental sub-system: Memory Management System. We have already talked about the main role which concerns the Bootmem in order to setup the Page Table allowing so the virtual to physical mapping of the kernel memory. At the startup, since there is just one thread running and since it must be as fast as possible, the data structures related to the memory management are quite simple and not very efficient. But now on it no longer the case, we have to discover all over the physical available memory cause allocations/deallocations would become very frequent and concurrent. We have also another problem related to the configuration of our machine: NUMA Nodes Organization. Since kernel linux must be NUMA Aware, it keeps and organizes memory relying on specific data structures like the struct PGLIST_DATA (typedef'd on the latter struct into PG_DATA_T) which represents a node. In case the machine is UMA organized the kernel linux creates just one node. All the nodes are linked in a NULL-terminated list called PGDAT-LIST. Each node is linked to the next via PG_DATA_T->NODE_NEXT. During the time a lot of code was re factored cause accessing into a global list of nodes was too costly, so it was transform into a macros which thanks to the node id, it can access all the nodes stored in a vector of pointer called *NODE_DATA[]. There were introduced a lot of facilities in terms of macros in order to work like an higher language. For example there is a macro called for_each_online_pgdat(), or first_online_pgdat(), or next_online_pgdat(pgdat). This is the best approach while programming: in this case others kernel programmers don't know about the real implementation of the PGDAT struct, their view is abstracted, moreover a change in the implementation of the struct will be translated only in change of the associated macros, leaving unchanged the rest of the code.

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;
```

A NUMA node is described, as we said, by the previous struct. Each node (portion of physical memory) is for its own divided into zones, this mechanism allows the kernel to differentiate the kind of usage of different portion of memory. Each zone is handled in different way through different allocators. Then there is a struct PAGE which represents like as a core map the physical memory available in a specific NUMA node, its size is stored into the node_size parameter. In the 32 bit scheme we have 3 different zones: ZONE_DMA (0), ZONE_NORMAL (1), ZONE_HIGHMEM

(2). In 64 bit system we have a couple of zone to add, but the most important are that 3 previously described. The first one is the DMA, typically it is small (less than 16 MB), this is a portion of physical memory whose purpose is to be accessible by the DMAC in order to communicate with I/O devices (eventually reserved to it). The second zone is the NORMAL memory. It is dedicated to the kernel and set up at startup by the bootmem for each NUMA Node. It is contiguous and on 32 bit system is up to 896 MB (it was changed a lot during time). So it is a limited area of memory which is used by linux to serve internal memory requests, it keeps this area always mapped in a virtual to physical way, and it is the only area in which we have a linear mapping from virtual to physical addresses (relying to va() and pa() macros). Since this portion of memory has all these features there is an high contention between the kernel subsystems, so the allocator must be as fast as possible to solve memory requests. The third zone is the HIGHMEM. Since the NORMAL zone is limited, kernel can use the remaining of physical memory of the HIGHMEM zone. This is not always mapped in Page Table, so every time the kernel has to travel the MMU scheme (also TLB) in order to let a portion of memory be available. So access this portion of memory is very costly and from here on there will be also the user space memory (malloc'd).

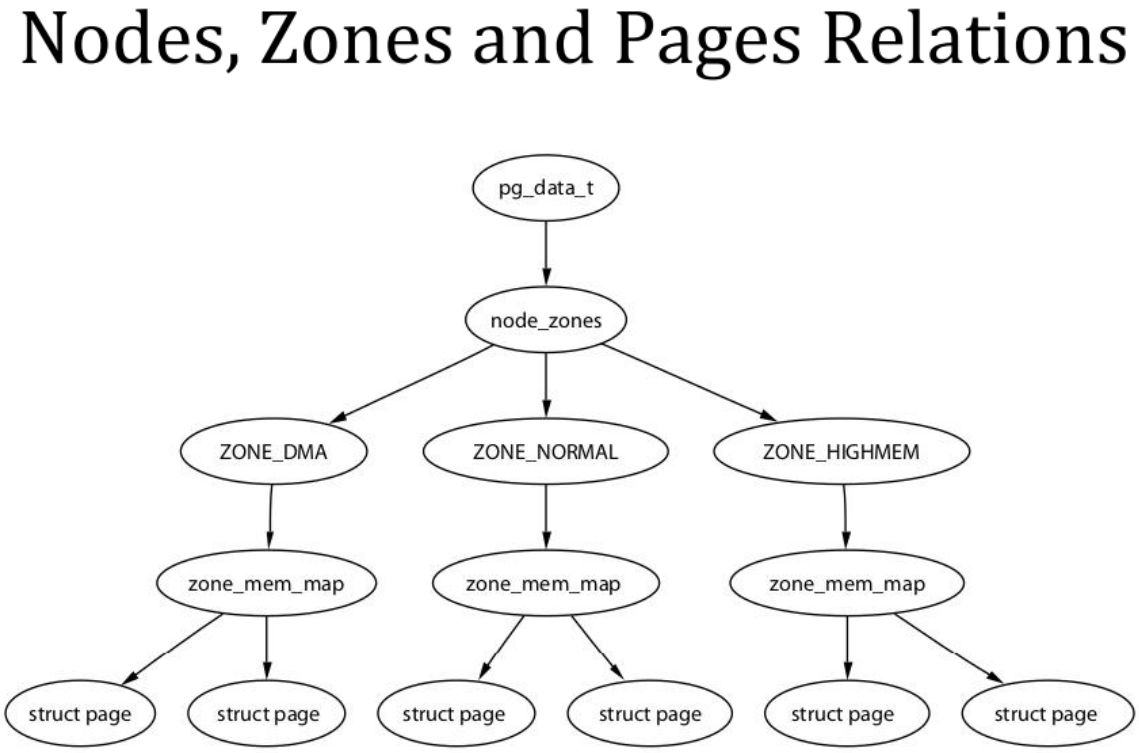
Zones are initialized after the kernel page tables have been fully set by paging_init(). It relies on the free_area_init() (UMA) and free_area_init_node() (NUMA) API. they grounds on Page Frame Numbers cause since the size of a frame is 4 KB, it is possible to find the physical address multiplying the PFN for the size of the Frame. The BIOS tells to the kernel through the e820 table what are the available frames of memory through their PFNs.

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 0x00000000000f0000-0x00000000000fffff] reserved
[0.000000] BIOS-e820: [mem 0x0000000000100000-0x000000007dc08bff] usable
[0.000000] BIOS-e820: [mem 0x000000007dc08c00-0x000000007dc5cbff] ACPI NVS
[0.000000] BIOS-e820: [mem 0x000000007dc5cc00-0x000000007dc5ebff] ACPI data
[0.000000] BIOS-e820: [mem 0x000000007dc5ec00-0x000000007fffffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000e0000000-0x00000000efffffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000fec00000-0x00000000fed003ff] reserved
[0.000000] BIOS-e820: [mem 0x00000000fed20000-0x00000000fed9ffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000fee00000-0x00000000feefffff] reserved
[0.000000] BIOS-e820: [mem 0x00000000ffb00000-0x00000000ffffffff] reserved
```

This is a dump example for the e820 table of the BIOS. It is built in the very early step of boot. It gives a lot of information related to memory mapping to the kernel. The first range of memory is the one used for storing data structures for the Real Mode of the kernel (such as the IVT), when it passes to the Protected Mode it is freed and so becomes usable. Then there some space reserver in which the BIOS maps itself during the boot sequence. Then there is a lot of memory usable from the OS. Finally we have some regions of memory in which are stored very useful data structures populated by the BIOS associated to the hardware available. Here the kernel retrieves information in order to initialize correctly the NUMA nodes.

A zone is described via a ZONE_T data structure. It contains two fundamental informations in order to manage it: the FREE_AREA_T struct and the PAGE struct. The page struct is used to describe the amount of physical memory available, kernel linux uses a lot of space to keep information about it in order to be as fast as possible to serve memory requests. The free_area_t implements the so called BUDDY SYSTEM which is an allocator. For a given buddy system it is associated a max order number (currently 11, they are the possible size of a buddy).



So recap: A NUMA node maps a piece of physical memory and it is described by a pg_data_t struct. Each NUMA node organizes its associated memory in zones, each one for a different purpose. Each zone is described by a struct called zone_t in which is kept a pointer to a page useful to store information on the available memory (pointers to other page structs which describe a PFN) and the reference to a private buddy allocator which manages allocations/deallocations in that zone.

Why the zone_mem_map (the page struct in the zone_t struct) is a page struct itself? Well the struct page in Linux contains a reference to a list_head struct. This is the way to implement in C the so called generic List<T>. So the mem_map_t is just a page itself that contains a reference to the first pointer to the pages list of a specific zone! In the page struct we have also an atomic_t count which represents the number of users actually using this page, it will be accessed only via RMW operations, cause it is declared atomic_t. Then we have flags which represent lot of information about the page, those flags could be reflected also in the page table scheme. Examples of flags are PG_locked, PG_referenced, PG_uptodate, PG_dirty, PG_lru, PG_reserved. How to manage flags? The linux kernel code is full of macros which are useful to SET, TEST and CLEAR a specific flag! The array that contains the MEM_MAP_T structures is called Core Map and it is kept in ZONE_NORMAL.

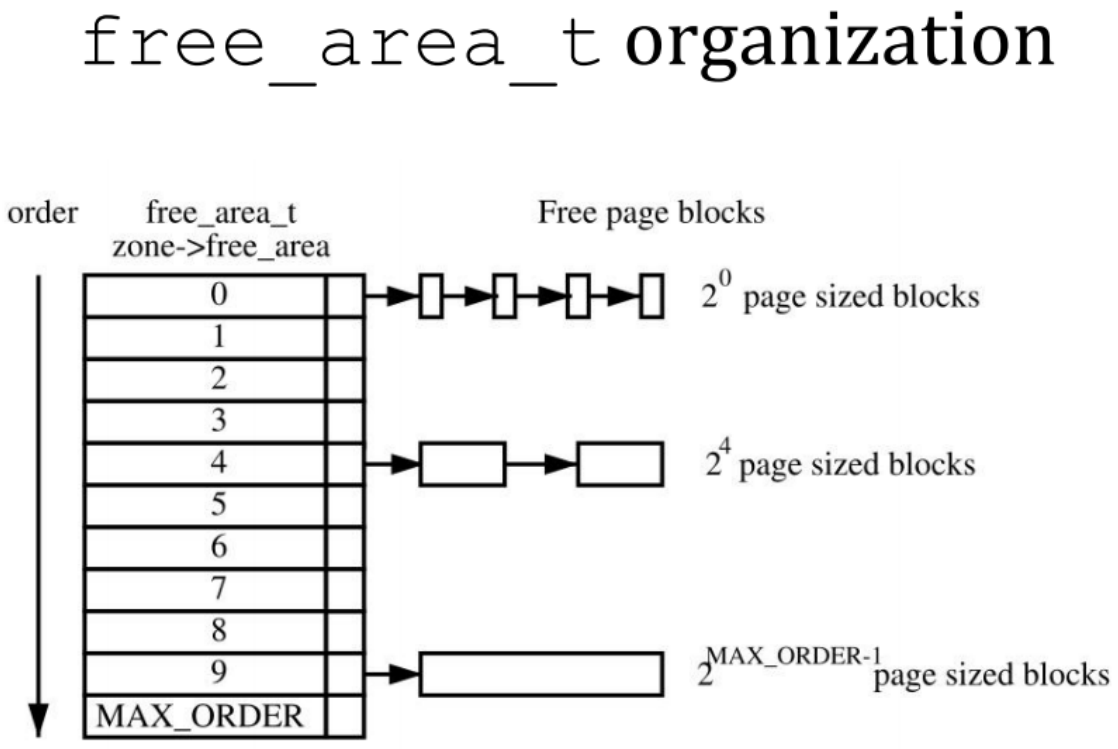
The Core Map on UMA systems is initialized via a single pointer MEM_MAP used directly in the free_area_init() routine. This routine creates a page struct for each PFN available in physical memory discovered by the bootmem allocator. Pay attention because now there could be a race condition on memory. Concurrently the bootmem allocator is still available and since some kernel subsystems can be set up concurrently, it must be avoided to serve some memory requests from them with an allocator and some others with the new one, cause this could be translated in a race condition on a same page of memory! So the routine of memory initialization

will store the value 0 in the count field of each page struct (noone is using it) but with the flag PG_reserved up. It means that no allocator will give back that page in order to serve a memory request. After all the __init memory is taken back and so the bootmem allocator deleted, the routine will un-reserve memory by resetting the PG_reserved bit. On NUMA systems we don't have any more a global mem_map array but each node (pg_data_t) takes its own pointer to the page list in the physical memory of that node.

The smallest granularity of a page that the buddy system can allocate to serve a memory request is 4 KB! So less than that can't be give back to a process. What about if I need more space than 4 KB? Well the buddy allocator divided a contiguous space of memory in 4 KB chunks. At each chunk is associated only one buddy (contiguous). It means that to serve memory requests greater than 4 KB I have to rely on buddies chunks! It means that if there is no buddy available I can't serve that memory request also if in total I have an amount of free space greater or equal than the requested one, but it has to be a buddy space (again contiguous in terms of buddy organization)! Ideally the buddy system is really fast in order to serve memory requests, but the external fragmentation is a real problem for this kind of logical-tree-based-view allocator. Since the minimum amount of memory which a buddy allocator can allocate is 4 KB, if the request is less than that value, there will be also a lot of internal fragmentation. It is obv a trade off between fragmentation and speed.

Memory Management - (27-03-2020)

The lowest level allocator is based on a buddy system installed in each zone of each node. Its best quality is experimentally proved the speed of serving memory requests but it suffers of external fragmentation. The linux kernel will set up others subsystem on top of the buddy allocator trying to solve this problem. The fundamental data structure which implements the buddy system is called free_area_t. It is composed by a free list of pages or group of pages and a bitmap in which is stored the actual allocation state of the ruled memory. Remember a frame has only one buddy!



At each zone of a NUMA node is associated a vector called free_area of strictures of type free_area_t. The number of element are equal to the number of orders or level in the buddy system. At each level we have a pointer to a free list of page blocks. Why page blocks? Cause we are talking about contiguous memory pages of different size, depending to the level we are

interested in. Obv at the max order level we are considering the total amount of memory managed by the buddy system and so it makes sense talk about page blocks (in this case just one i.e. the entire memory managed) of size 2^MAX_ORDER. This reason can be applied at each level. The level that would be chosen is related to the memory that is requested. Obv the level 0 is associated to a request of memory less or equal to a single 4 KB page. Since the fact that the memory possibly at the start time is quite completely free, the buddy system will be initialized only with few entries in the free lists of the highest levels, cause we are considering the total contiguous memory available. So if for example we have the total amount of memory free, we will have only an entry of size 2^MAX_ORDER in the free list associated. If the first request is up to 4 KB it is interesting to analyze the split process that is made recursively: at the first time the only block available is split into two buddies, each one with a size which is the half of the father's, then it is chosen one of the two buddy and then this process is repeated until the requested size is reached (in our example a single page). Notice that after the split we will have an element in each leveled list due to the continuous splitting. When a chunk of memory is freed, the buddy system will check if its buddy is used or not. If it is used the freed chunk will be put in the associated list, otherwise it will be coalesced with its buddy. Also this reason is made recursively until it is reached a busy buddy or the max order. The buddy system in linux is protected by a spin lock from race condition while two thread are requesting memory contemporaneously. The execution so might become slower due to the spin lock. The metadata associated to the state of memory is just composed by a bitmap. The bitmap uses a bit for each pair of buddies. It's a fragmentation bit it means that if it is 1 just one buddy is actually used, otherwise both buddies are free or are currently used. How can we distinguish that? It depends on the operation that we are trying to execute. The buddy system, as we already said, can be seen logically as a tree (it is not actually implemented like a tree). So the bitmap is organized taking into account the level at which it refers and the position of the node at a specific level of the tree. In this way the access in the bitmap is immediately performed through simple math operations.

After we have populated this free_area struct for all zones associated to each NUMA node, we are ready to destroy the bootmem allocator as we said so far via the mem_init() routine. Now the buddy system is able to serve memory requests and we can release all the previously blocked memory (think about the PG_RESERVED bit trick).

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;
}
```

We have to cycle into the bitmap associated to the bootmem allocator. We check if the bit is used or not. In the case is it not used we have to reset the PG_RESERVED bit and we fake the actual buddy system into thinking it is a real free increasing by one the count number (number of references to that page). In this way the buddy system will perform a regular free operation via the __free_page(page) routine and it will give that memory to the ZONE_NORMAL. At this point all the memory is managed by the buddy system.

Now, we have to differentiate between the different allocation contexts. The first one is called Process Context, it is due to a system call. We have always think about the worst scenario in which we don't have enough physical memory to serve the request. It applies a Priority-based approach, it means that an user request has a low priority and in the worst case it must wait along the current execution trace (system slow down). A completely different context is the one associated with Interrupts. While we are executing an interrupt handler (due for example to an activated driver) we cannot wait for memory, cause maybe we are running with IF disabled, so we must be as fast as possible in order to make the entire system responsive. So it is applied an independent priority scheme and if a memory request cannot be served it should fail. So different situations means different allocation contexts and finally different policies.

Now let's look to the internal MM API for the buddy system. Since the buddy system works with a 4 KB page granularity, also the API reflects this size. The first one is GET_ZEROED_PAGE(int flags), it returns the virtual address of the page requested and the page will be filled with all zeros. The page is chosen depending on the NUMA nodes policy, cause given a node, the buddy system associated to it will return the requested page. The same thing without filling with all zeros does the __GET_FREE_PAGE(int flags). It is called also from the get_zeroed_page. If we want a bigger block of pages we can rely on the __GET_FREE_PAGES(int flags, ul order). Depending on the order we can obtain a block of 2^order pages. In the very end this is the only entry point for asking memory to the buddy allocator cause __get_free_page calls __get_free_pages with order set to 0. The corresponding free API are: in case of freeing just on page we can rely on free_page(ul addr), we have to pass the virtual address which, thanks to the virtual to physical mapping, will be translated to a PFN and then put again in the corresponding free list or coalesced with its buddy if this one is free too. If we want to free a block composed with a greater than 1 number of pages we can rely on free_pages(ul addr, ul order). The operation is the same, in fact free_page calls free_pages with order equal to 0. In order to be as fast as possible kernel linux doesn't implement some sanity checks in order to not corrupt the buddy system. Since the buddy system is directly used from a small number of subsystems, the checks must be done at caller level! Pay attention to pass the right parameters to the buddy's API. They are low level API, if you use it, u must know what are you doing. The FLAGS used in the API specifies the allocation context: we can use GFP_ATOMIC (get free page atomic) to tells the buddy system that we are running in an Interrupt context and so this memory request cannot lead to sleep, the remaining three flags (GFP_USER, GFP_BUFFER, GFP_KERNEL) are all associated to User context and can lead to sleep if needed. In the modern x64 systems these priorities are considered as the same, cause the amount of physical memory available is huge, and it is rare that a memory request cannot be served. Instead in the previous systems, smaller and slower, it was a good practice to differentiate between the likelihood to lead to sleep. The GFP_USER means that we are allocating memory for userspace-related activities but it is still a kernel memory!!! It is not an user space memory, it is only related to user behaves. GFP_BUFFER tells that we are allocating a buffer, for example one to read and write on disk. GFP_KERNEL tells that the requested memory is for kernel operations. Obv the latter is the fastest as to be served.

On NUMA systems we have multiple nodes in which we can take a single page to serve a memory request. Obv an UMA system invokes NUMA API but the system is configured to have a SINGLE NODE. So the entry point described so far __get_free_pages() will call the real buddy API specifying also the NUMA node id! This function is called ALLOC_PAGES_NODE(int nid, ui flags, ui

order). The nid, if it isn't set manually from an user, is chosen according the NUMA policy used. Since Kernel 2.6.18, userspace can tell the Kernel what policy to use. We have 4 different policies: the MPOL_DEFAULT which will rely on the NUMA node associated to the core that is actually requesting the memory, then we have the MPOL_INTERLEAVE policy which implements a round robin algorithm between NUMA nodes, then there is the MPOL_PREFERRED which tells a favorite NUMA node, but if there is no space we can receive memory from a different NUMA node, then there is MPOL_BIND the same as PREFERRED but in case of no memory available the request fails. In order to apply our chosen policy we can rely on the SET_MEMPOLICY function (at the end a syscall) specifying the policy and also a mask of nodes that must be set. The MBIND function instead can also specify the region of memory in which apply the requested policy passing a base virtual address and an ul len that tells how much bytes is long that region of memory. We can also move pages (MOVE_PAGES) from a node to another specifying the pid of the process, the pages that must be moved and the node in which we want to move them. In this case multiple buddy allocators must be modified accordingly.

Now let's analyze the HIGH MEMORY ZONE. Remember that the NORMAL ZONE is the Kernel stuff one. Here there are all the memory related to the unzip routine made at startup time and other structures that reference directly normal zone. It is ALWAYS MAPPED in page table and this is an interesting aspect in terms of performance: if the kernel deallocates some pages in normal zone, in has to manage that only via software cause in the firmware will be never deleted that translation from the page table, and so there will be never need to flush the TLB! This is not the case of the HIGH MEMORY ZONE. Every time we allocate some memory in this zone we have to explicitly construct a the corresponding translation in the page table. And this means that we have to update the TLB in order to make it consistent with our operations (for this core or maybe all of them!). High memory is really important in that situations in which the virtual address space is smaller than the physical memory available. Think about a 32 bit system (with a virtual address space up to 4 GB) with a 16 GB of physical memory. How can we use all that memory? Thanks to the high memory which is not covered by a PERMANENT translation in the page table. This zone is the one where userspace memory comes from. In the most recent 64 bit system, since we have a virtual space much larger than the 32 bit one, also the high memory is managed with a sort of linear mapping in order to speed up allocations. Since Linux wants to be as portable as it can, it still keeps the high memory zone managed as we said.

The API used to interact with high memory are: VMAP() is used to make a LONG-DURATION MAPPING of multiple pages (notice that in the virtual space the addresses will be allocated contiguously), KMAP() permits a SHORT-DURATION MAPPING OF A SINGLE PAGE, it also needs global sync of page tables but that cost is amortized somewhat due to the short duration of the mapping, then we have KMAP_ATOMIC() which permits a very short duration mapping of a single page restricted to the CPU that issued it (it means that only the local page table is updated, if the process is scheduled on another core it will segfault). This latter API is useful if we are handling a driver: if memory in zone normal is not available and since we want to be fast we can ask memory from highmem without the constraint to perform a global synchronization. The problem during deallocation is that we can have a physical page virtual mapped to two different region of virtual space. How can we solve this problem during deallocation? We have to keep track of who is actually referencing a single PFN. The kernel keeps an array of counters called PKMAP_COUNT[], a count for each high memory page. A counter could be 0 if the page is not mapped, 1 if it is used to be mapped, or greater than one which means that the page was mapped n-1 times. The case in which a counter is equal to 1 is a special case due to the KUNMPA() operation. This routine decrements the associated reference counter, when it is set to 1 means that the last user has destroyed the translation of the virtual address but the CPU still has cached that mapping. This mapping has to be invalidated relying on for example a __flush_tlb_all(). Why this? Cause our thread of execution is scheduled among different CPUs and it required the access to that page in each CPU in which he is scheduled. So different CPUs could

cache the translation in its own TLB. After flushes all the TLBs we can set the reference counter to 0.

As we said before buddy allocator is not the right one to serve fast allocations/deallocations cause it is affected to external fragmentation and it is ruled by a spin lock which could be make slow the performance of the buddy allocator. For example every time we want to allocate or deallocate structures related to the page table scheme (pgd_alloc() or pgd_free() for example) , Linux kernel relies on on of the Kernel-level FAST ALLOCATORS. For paging are used QUICKLISTS allocators, they allow us to get single free page used typically for building virtual to physical translation. For other buffers the kernel uses SLAB allocators, which are lists of predefined size buffers available. Every time a memory request is issued the first one which will try to serve that request is the SLAB, if it doesn't have enough memory it will rely on the buddy system. There are 3 kinds of different allocators: SLAB, SLUB, SLOB.

QUICKLISTS are list of free single pages, they are implemented as a per-core page lists (no critical section cause its a personal list!). It means that the core which is requiring memory to set up a translation from virtual to physical memory will control its own quicklist to retrieve that memory first. It is very fast because there is no need for synchronization. If allocation fails they rely on __get_free_page() (buddy system).

Quicklist Allocation

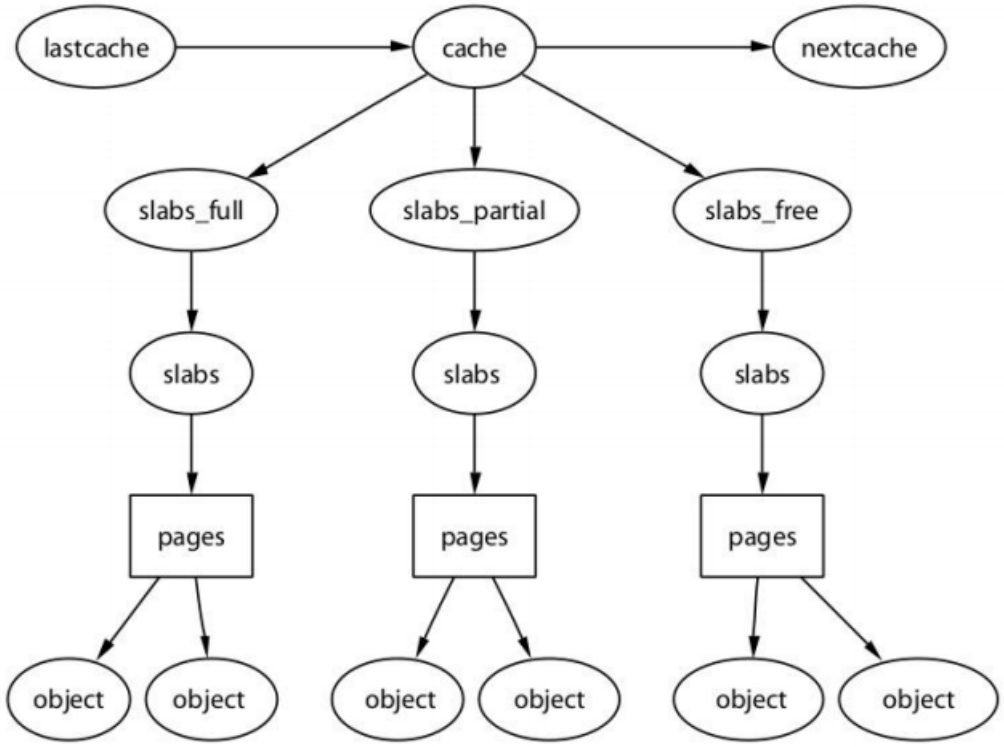
```
static inline void *quicklist_alloc(int nr, gfp_t flags, ...) {
        struct quicklist *q;
        void **p = NULL;

        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;
}
```

A single CPU can have more than one quicklist, for example one for each level of the page table. It has to be specified (nr). Then we have to specify the GFP_T flags in case we will rely on the buddy system, if it is the case we will ask also via the GFP_ZERO flag to zero the content of that page. The quicklist struct (array of pointer) is retrieved relying on an internal function that maps using a same name the actual struct which belongs to this current CPU. How can this per cpu approach be possible? We know that each CPU is associated to a portion of virtual addresses. So there is a sort of global array that for each core id stores the base virtual address of that portion of memory in which there are those per cpu variables. Thanks to name of the variable which we are trying to access in the compiler can translate it as an offset. But this operation is too costly cause the cpu id is discovered relying on the CPUID asm instruction which performs as a memory barrier (it means that it will empty the store buffer in the CPU) and the memory access is done via an "huge" number of asm instructions. So it was thinking to use the gs register to keep track of the base address of this portion of virtual memory in order to be as fast as possible. Different gs

register will point to the same entry in the GDT, but since we have a different GDT for each core, we can store a different value depending on the core to which it belongs. Notice that for this approach we are using segment in a no flat mode cause the base address is different from 0 (it is different for each core). After we have retrieved the pointer to a specific quicklist we can search for the first valid page cached in it. If there is no page available we simply rely on the buddy system API at the end of the routine, otherwise we have to reason about the way to store the list of pages. Notice that the p variable is declared as a pointer to pointer, why? Cause a page is 4 KB size but in order to save space and not use another header structure we use the first amount of byte of the page to store the pointer of the next page. This list is a simple linked list NULL-terminated. So when we assign q->page to p we are pointing to a pointer, the next one! So we have to unchain the first page, we have to set the first free page of the quicklist to the next of the selected one (q->page = p[0]). Then we clean the pointer stored in the page and then decrement the number of pages available in the quicklist. At the end we can return the pointer found as a void* casted one. Why put_cpu_var call? Cause the kernel code can be preemptable! If we are descheduled after accessing a per cpu variable of core 0 and the we are rescheduled on the core 1 with the instruction "q->page" we will accessing that variable not from the owner core! We will break the isolation purpose of the per cpu variables. So in the get_cpu_var the linux kernel will disable preemption in order to be safe about this kind of problem. After all the operation on that variable with the call to put_cpu_var it will enable preemption again. The other check between return the pointer is done because the function is called with preemption enable and it has to leave with preemption enable! So before return we have to re enable preemption, then check if the code executed with preemption disabled gave us a valid pointer and just now we can return. What is the cost of the added check? Quite none cause we use again the likely directive which is translate in a __builtin_expect(!!(x), 1) (otherwise 0 for unlikely resp.). This is a gcc directive which will reorder the execution path in order to prefetch the most likely instruction directly into the pipeline. The double !! transform everything in 1 or 0 (ex. a pointer != null is 0xXX...XXX first is transformed in 0x00...000 and then in 0x00...001).

Let's analyze the SLAB allocator. It works on group of SLAB. It can be seen as an object (a struct in os world) which tries to reuse memory, a sort of cache of what is done until now. It is a set of caches in which each cache tries to serve memory requests of a fixed size. The size can be smaller equal or greater than a 4 KB page. It tries to reduce also internal fragmentation.

The SLAB Allocator



We have three different groups. The first one is the SLABS_FULL: here are stored all the pages, so the object in it, that are completely full, it means that actually all these objects are used by someone. The second one SLABS_PARTIAL is the first attempt in order to serve a memory request cause here are store all the partial pages in which some object are currently used and some others are free. If the last free object in a page is given to someone for some purpose the page is moved in the first group, if one of the object of a page is freed and previously the page was in the first group, it is moved to the partial group. The third group instead SLABS_FREE is composed of pages whose objects are all actually freed. If an object is allocated the page is moved to the partial group, if a partial page is completed freed it is moved to the free group. This cache will give back pages to the buddy system if and only if the number of free pages in the third group is very large (over a threshold). This is also the case in which there is a out of memory error during allocation process, this will trigger a forced release of memory cached in the SLAB allocator. But in the modern systems with an huge amount of physical memory this case is very rare and the SLAB is a really good cache to avoid requests to go down to the buddy system increasing overall speedup. If the we have to serve a memory request and we have the partial and the free groups empty, we rely on the buddy system of course. This one will give use a number of pages equal to 2^order, where order is specified by the SLAB. So this kind of memory is contiguous! This portion of memory, virtually mapped of course, could be split in multiple object depending to the memory size requested to the SLAB. Remember that different caches handle different object size, this is the reason between the linked list of caches that composed the SLAB Allocator.

So this is the most used interface for memory allocations and deallocations cause it abstracts the limitation of the buddy system and prevents wrong parameters to be passed to the buddy system APIs. So for allocations we can rely on the KMALLOC(size_t size, int flags) function. The kmalloc is very similar to the malloc facility for userspace application but it allocates memory for kernel. It takes a size which tells how many bytes we want to allocate. Of course that size will map our object in the corresponding cache of the SLAB that manages object of that size (or the immediately greater). We also specify some flags, used for the GFP flags because in the worst case we want have to revert on the buddy system. The common use is that the kmalloc function

returns memory from ZONE NORMAL, the memory in fact is contiguous and we have a always set mapping for this zone. The counter part is the kfree(void *obj) function. It takes the pointer to an object and it will put that object first of all in the partial group. If we want to perform a NUMA-aware allocation we can rely on the kmalloc_node in which we can specify also the node id in which we would perform the allocation.

Available Caches (up to 3.9.11)

```
struct cache_sizes {
     size_t                cs_size;
     struct kmem_cache     *cs_cachep;
#ifdef CONFIG_ZONE_DMA
     struct kmem_cache     *cs_dmacachep;
#endif
}

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

For each cache we have a struct cache_sizes in which there is the size managed by the actual cache of course, a pointer to the SLAB for the zone normal and one for the ZONE DMA possibly. Then we have a global array of these caches, one for each power of two size. Every time a memory request is issued, the memory size is rounded up to the nearest power of two, and if the SLAB associated is NULL yet, it is set up and used (Lazy approach of the Linux Kernel).

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
};
```

Later on the struct was change in order to explicitly represent the 3 different groups in a cache. We also have a spin lock which is taken every time we have to move pages around the different lists. There is also a new concept which is represented by the colour_next variable. We said that the SLAB allocator wants to deal also with internal fragmentation. But if we simply divided in objects a page we kill the performance from a cache point of view. Remember the FALSE CACHE SHARING problem in which two different variables of the same struct are actually used from two different threads of execution, in a "isolated" mode, it means that the entire struct is shared among these two threads but each thread uses only one item of that struct. If we don't do nothing for solve this problem, every time a thread wants to access that variable it has to load in cache also the near variable used by the other thread, and vice versa. This kind of ping-pong due to the cache subsystem of the firmware it a performance killer. So at compile time we can ask to the compiler to add some padding in the middle in order to align these two variables to a cache line, solving so the problem. Each thread would cache only the correct variable and some useless padding, never asking for other cache line. BUT this good approach would be killed if we not take in account it during the objects allocation. Of course the first object in a page is cache aligned cause it start at the beginning of the page, and so the padding added from the compiler prevents the false cache sharing problem. But for the rest of the allocated objects (Immediately taken in a contiguous way) this is not true, it means that hey are not cache aligned. So the colour_next variable represents the padding in bytes added after an object in order to keep objects cache aligned in a sub page context or a multiple pages one too.

The other problem kernel linux has to deal with is the one of Aliasing in cache subsystem. Some times we need to make a completely or a partial manual cache flush operation. To do that we can rely on some APIs very similar to the TLB ones. We have flush_cache_all which flushes all the CPUs cache systems (maybe due to a kernel page table change), it is very expensive in terms of performance. Or we want to flush a portion of memory related to the userspace application and to do it we can use flush_cache_mm(struct mm_struct *mm), or just a range of addresses respect to userspace application using flush_cache_range passing also a start and an end pointer. Finally we can flush only a page via flush_cache_page, it takes a vm_area_struct as input which is related to the mm_struct previously analyzed, in fact we can access one from the other cause they both represent a portion of memory related to userspace application, but in this case we can also test

if this portion of memory (page) is executable or not, cause it is really important to know if this info have been loaded in a instructions or data cache (a lot of architectures have different caches for different purposes).

Memory Management - (01-04-2020)

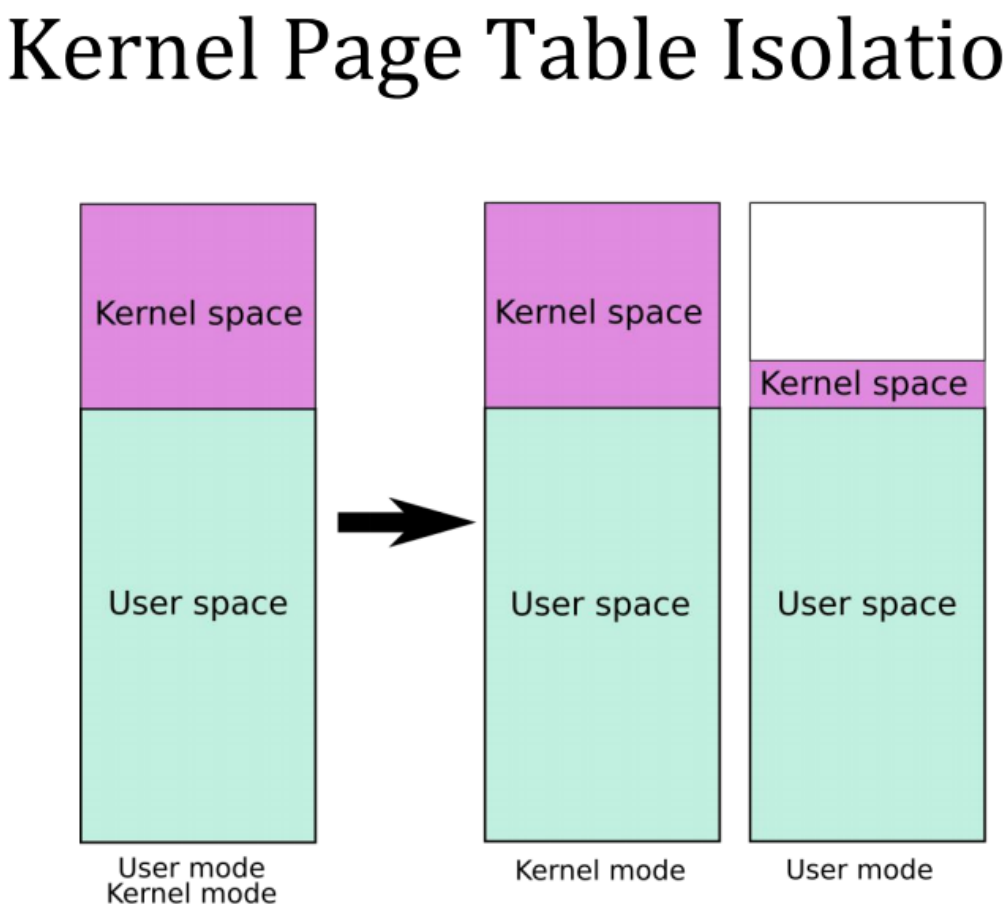
The last thing to discuss is the high level API for the high memory. Remember that the low level APIs are KMAP() and VMAP(). When kernel wants to make a large size allocation, it relies on high memory. Why this? Cause zone normal is small wrt the size and cause if we want a large portion of memory usually we don't want a permanent virtual to physical translation. This is the case of mounting external modules. So we want a stable way to retrieve memory but this can be release whenever we want, this is the purpose of the low level function VMAP(). The high level API for this purpose is called VMALLOC(ul size). We can ask for any size we desire but since it relies on high memory it could be that the memory is taken in a fragmented way, but contiguous in the virtual address space. The counter part function is VFREE(void *addr) that frees the previously taken memory.

But how can we translate virtual addresses into physical ones if they are pointers to high memory zone? Cause we can have virtual contiguous addresses but totally different physical pages wrt to position. We cannot apply __pa() and __va() because they exploit the trick of the constant (random) offset between virtual and physical addresses for kernel data and code in zone normal. In order to use FOR ONLY ZONE NORMAL POINTERS these macros we are encouraged to use virt_to_phys(ui addr) and phys_to_virt(ui addr) cause in the very end they will call that macros. These facilities are not going to change during time but we cannot say the same for pa() and va().

So KMALLOC vs VMALLOC. The kmalloc can manage only memory requests up to 8 KB, due to the false cache sharing problem. If the Request is greater it actually fails. Vmalloc can serve up to 64/128 MB of memory cause it is taken in high memory. Kmalloc gives contiguous physical memory cause it relies on the buddy system, vmalloc not. Kmalloc exploit permanent mapping of zone normal so the TLB is already consistent on all the CPUs of the system, not for vmalloc that every time it must perform allocation on page tables (sync TLBs on different CPUs) in order to set up the correct v2p translation.

How can we move memory from and to kernel space and user space? There are a lot of APIs available. Why they are so important? Let's remember the division of virtual space and let's analyze the 32 bit view. In the last 1 GB of virtual memory we have the virtual memory for kernel space: we have both the zone normal data permanently mapped in the page table and the vmalloc'd memory already mapped in it before being allocated. So in the SAME page table there are both the v2p mapping of kernel and userspace. Thanks to USER/SUPERVISOR bit in page table we can prevent user to read and write kernel memory cause it is running at R3, but kernel in principle can do whatever it wants on user space memory. If a new process is running, CR3 will point to a new page table consistent with the new process memory but it will keep always information at least for zone normal of kernel memory. So if the kernel can in principle access to user space memory why do we need APIs to do that? Cause kernel must not believe parameters passing by the user, due to bugs or malicious actions. If a segfault happens during a syscall, the kernel will crash! So all those API rely on a simple function which performs a kind of security check on the memory region that the user wants to access. It is called ACCESS_OK and it checks the correct mapping of an address for a certain size of bytes from it comparing it with the mm_struct that describes the actual memory context of an userspace application (we will see it later). If the access is ok it returns a non zero value (true). So our APIs will rely on that function in order to safely access userspace memory. We can COPY_FROM_USER(void *to, const void *from, ul n), COPY_TO_USER(void *to, const void *from, ul n), GET_USER(void *to, void *from) or PUT_USER(void *from, void *to) (for integers), STRNCPY_FROM_USER (char *dst, const char *src, long count).

Now consider this permanent mapping of kernel memory due to meltdown. In this way when we try to load some kernel data it will actually loaded in cache, then the MMU will trigger a segfault which we can capture in order to make happy the firmware and then perform a timing attack on cache accessing cause the kernel data was already loaded concurrently with the security check performed by the MMU looking at the SUPERVISOR bit of the page table entries. The problem is that the v2p translation of kernel memory is always valid in the page table. The only point available in order to solve this problem via software is in the Page Table. We have to change the way in which CR3 point to different Page tables depending on the Ring in which we are actually running. The patch is known as Kernel Page Table Isolation.



To the left is the traditional view of virtual address space when kernel memory is always mapped. To the right the patched one in which when we are running into user mode only few kernel memory (GDT, IDT, gate code) is actually mapped in order to still let user application transition into kernel mode. When we are running in kernel mode the page table is changed according to map all kernel memory correctly. In this way when an user tries to access kernel memory the v2p translation will crash it in order to avoid cache side effects. But obv changing CR3 is not enough due to the TLB that caches the v2p translation of kernel space, but this is not a problem on Intel CPU cause changing CR3 means flushing TLB via firmware. This is very critical wrt performance for system calls intensive application like the DBMS in which we have observed a lose of performance up to 70%. The minimum amount of data that in user mode are still available are: the gdt of course, an entry_stack_page to perform the initial part of the transition from user mode to kernel mode, the tss, and the stacks used for managing interrupts (they can arrive during the user mode and we have to manage them accordingly). Since we have more than one CPU that run different processes, we can't have a single cpu_entry_area structure, so it is declare as a per cpu variable! How can we do this efficiently? We know that all the memory which is related to kernel is contained in a single entry of PML4 level (the most external one) in the page table scheme. It means that in order to hide kernel mapping in user mode we can just null a single entry of that table. It is enough to store two different PML4 tables (one with an one without

kernel mapping) and point the right one through changing the content of CR3. All the remaining stuff of the page table will remain the same and can be shared among the two different modes (we not duplicate the rest of page table level entries). The memory wasted is 4 KB, the size of a PML4 entry. Moreover these two PML4 entries are contiguous, it means that the offset between them is just 4 KB. We can do that flipping just 1 bit!

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
```

The first thing to do is to update the content of CR3. This is done relying on the SWITCH_TO_KERNEL_CR3 MACRO that takes a scratch register to do the job, it means a register that it will be free to be used at that point of the code. The first operation is to move the content of CR3 in the scratch register, then we perform a bitwise operation to add just the 4 KB bit (bit 12), once we we have set that bit we can save the value in the CR3 register. Note that flip a bit means that the original one must be 0 initially! This is the reason between the limitation of the random position of kernel stuff due to KASRL. At this point the flush of the TLB will be triggered. At the end we can do the inverse operation and then rely on the macro SWITCH_TO_USER_CR3_STACK.

What happens if I do some memory allocation in user space that requires the allocation of a new PML4 entry? When we are asking for memory we move to kernel mode, then the allocation will be performed and the the KERNEL PML4 table will be updated with the new entry. But when I come back to user mode the user application will crash if I don't copy that "shared" entry also in User mode PML4 table. In principle each modification done in kernel mode must be copied in user mode PML4. This operation is called Memory View Consistency. To explain this we have to clarify how the kernel manage the user requests. We know that the kernel is lazy to serve the memory requests coming from the users cause it doesn't trust them. In fact if a malloc is issued and the mmap (the kernel level API for allocating memory for user applications) will not touch the page table! Because if it touches it it will reserve physical memory, but we are using a lazy approach. We believe that the uses will rely on a significantly smaller amount of memory wrt the requested one. The only thing that the kernel does is make VALID some portion of virtual address space, not allocates memory. The the user writes a byte starting from a virtual address, the write op is intercepted by the MMU and checking the Page Table it discovers that the v2p doesn't be already set up for this address and sends a trap to the SO, a memory fault. The fault will trigger the fault handler associated. The SO will take control since now and it will look into the mm_struct associated to that process, it will discover that the fault generated is a minor fault

cause the virtual address is valid but the v2p mapping hasn't still been performed. Just now the kernel will reserve physical memory for the application, a portion memory related to the actual bytes that user wants to write! Then the write operation is re executed and it will correctly performed. The same approach is used to recover the fault that will be generated if a user wants to access the portion of memory allocated only in the Kernel PML4, we can recover that fault. The fault that we want to recover is due to a vmalloc'd memory (cause it is not permanent mapped). First of all while executing the fault handler we are running in kernel mode, so we read the content of CR3 (that points to the PML4 of the kernel), we transform it into the corresponding virtual address via __va() cause it is in zone normal, then we add the offset which is related to the address that we want to map, we can do this cause the system is using the flat mode and the virtual address corresponds to the linear one and the we can take the highest bit of that address as the index of the entry in the PML4 table and some such index to the base pointer using the arithmetic of the pointers. The if the entry is currently null in the PML4 of the kernel we set a PGD entry passing that previously computed pointer. The set_pgd function relies on a WRITE_ONCE macro that allow only one write operation for that gdp entry also in a concurrent system relying on a RMW instruction. The value that it will write in the PML4 of the kernel is the result of a set_user_pgtbl function. This function first of all check if the entry that must write in the PML4 of the user actually maps to user space, if not the it returns, otherwise flipping the bit 12 it writes in the corresponding pgd entry of the user the entry passing as input. Then in order to avoid spectre attack it will force the PAGE_NX bit to be 1 (if supported in the actual architecture) in order to prevent user memory to be executable in the kernel memory. Then it return the pgd entry and it will be stored also in the kernel PML4.