1044414: Advanced Operating Systems and Virtualization **Spring 2018**

Lecture 6: March 20

Lecturer: Alessandro Pellegrini *Scribe: Anxhelo Xhebraj, Beatrice Bevilacqua*

6.1 Memory Management

As seen in the previous lecture there might be systems providing NUMA differently from Uniform Memory Access (UMA). Linux handles both cases similarly since UMA can be seen as a degenerative NUMA system having just one node.

Each node is described by `struct pglist_data` (typedefined to `pg_data_t`) even if the architecture is UMA. All the nodes structs are linked together forming a linked list called `pgdat_list`. In the UMA case only one `pg_data_t` structure called `contig_page_data` is used. From the kernel version 2.6.16 the `pgdat_list` has been replaced by a global array called `node_data[]` and the iteration over it is done through macros defined in `include/linux/mm/zone.h`.

Each node is divided into a number of blocks called *zones* which represent ranges in physical memory. A zone is described by `struct zone_struct` typedefined to `zone_t` namely `ZONE_DMA, ZONE_NORMAL, ZONE_HIGHMEM`.

`ZONE_DMA (0:16MB)` on x86 32 bit is associated with the first physical 16MB and is used for Direct Memory Access. This is done to remain compatible with constrained devices that are not capable to address more then 16MB. In Linux it is also for disk access (?).

`ZONE_NORMAL (16MB:896MB)` is the range of memory that is always mapped to the kernel's virtual memory.

`ZONE_HIGHMEM (896MB:End)` is present if there is more physical RAM than can be mapped into the kernel address space. Thus it is not directly mapped to the kernel's virtual address space, instead it is remapped whenever it is needed. In x86 64 bit this zone is not present since the kernel virtual address space is not confined to 1GB therefore all physical memory can be directly addressed by the kernel virtual space.

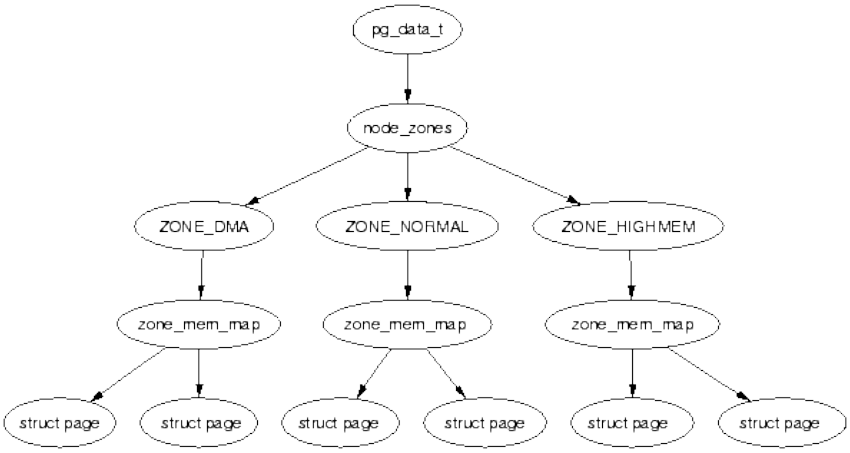

Figure 6.1: Relationships between structs. Note that there are multiple `pg_data_t` linked together.

`pgdat_list` is created incrementally at each `init_bootmem_core` call by prepending each `pg_data_t`

To each page frame in the system there is associated a `struct page` element that holds all the information needed by the kernel to manage that frame. All the structs are kept together in a global array called `mem_map`.

We are going to describe all the structs represented in Figure 1 starting from the top of the image.

```
> 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;
```

`node_zones[]` holds the `zone_t` structs for each zone.

`node_mem_map` is the pointer to the first `struct page` within `mem_map` that belongs to this node (all the other `struct page`s of the node follow contiguously in `mem_map`).

`node_size` is the total number of page frames belonging to the node.

`node_next` is the pointer to the next node in the list `pgdat_list`.

In the i386 architecture (in which just UMA is supported) the only node `contig_page_data` is initialized in `free_area_init()` (`mm/page_alloc.c`) and `zone_t` fields are filled thanks to the parameters discovered beforehand through the E820 facility passed to this function. In the NUMA case (64 bit) node initialization is done in `setup_arch()` which indirectly will call `free_area_init_node()` (`mm/numa.c`). Both functions (`free_area_init_x`) will eventually call `free_area_init_core()` (`mm/page_alloc.c`) that performs the setup of the data structures described below.

During the POST phase the BIOS discovers how much physical memory is available and setups a table called E820, which contains information about how much physical memory is available and which are the usable regions (for example in case of shadow-ram initialization the BIOS must inform the kernel that some portion of memory is not available since it stores BIOS routines).

```
> include/linux/mmzone.h
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;
```

`lock` is used to protect the zone from concurrent accesses.

`free_area[]` is an array of structs used for memory allocation by the Buddy System.

`zone_mem_map` similarly to `node_mem_map` points to the first `struct page` of `mem_map` that belongs to this zone.

```
> include/linux/mm.h
typedef struct page {
    struct list_head list;
    struct address_space *mapping;
    unsigned long index;
    struct page *next_hash;
    atomic_t count;
    unsigned long flags;
    struct list_head lru;
    struct page **prev_hash;

    #if defined(CONFIG_HIGHMEM) || defined(WANT_PAGE_VIRTUAL)
        void *virtual;
    #endif
} mem_map_t
```

`list` is a field used to link this `struct page` to other `struct pages` forming a list of pages satisfying a certain property (e.g. free pages, dirty pages, locked pages etc.).

`count` tells the number of processes that are using that page frame (). If it is 0 then it can be put into the list of free pages (usable).

`flags` describe the status of the frame.

`virtual` is used for pages in the highmem area. virtual then accepts the virtual address of the page if it is mapped to the kernel address space.

When pages are initialized in `free_area_init()` the count field is set to 0 and `PG_reserved` bit within the `flags` field is set to 1 so that no memory allocator except for bootmem (which doesn't rely on these data structures) can allocate that frame. This is done because the Main Memory subsystem of the kernel will not use the bootmem allocator anymore in its steady state but new kinds of allocators and therefore it must ensure that there aren't conflicts between the two.

Frame un-reserving is performed in `mem_init()` (`arch/i386/mm/init.c`) and it will allow the steady state allocator to start working.

6.2 Buddy System

Fast algorithm. Suffers of fragmentation. Recursively split and find the best fit for the requested memory.

There is an array of free_area_t where the pointer to the free pages is kept into list head. The integer is used as a smart bitmap. 1 bit for a pair of buddies. Any time that you allocate or free one of the two buddies you invert the buddies. 1 means only one of the two buddies is in use while 0 that either both are available or none.

The mem alloc starts from higher order to lower to find which buddy is best.

6.3 High Memory

vmap: long duration mapping for multiple physical pages.

kmap: short duration, needs global sync, all the cores running in kernel mode will see that mapping

kmap atomic: short duration but only this cpu will see that mapping, used in interrupt handlers usually.

Deallocation: vector of counters for each high memory page. 0 is not mapped, not used by anyone else. 1 just been released. more than 1 it is mapped multiple times. The release is done with kunmap that decrements the counter. When the counter goes to 1 means that no kernel thread is referencing that page.

On mapping we create a PTE.

All the CR3 registers point to the same page table since it is not useful to have different views on memory. When counter 1 the PTE is removed and the tlb is flushed to be sure that all the cores remove that from tlb.

The struct pages hold the physical addresses of the pages that are then translated to virtual ones.

Memory allocation is finalised with mem init. Destroys the bootmem allocator and releases all the frames used by bootmem through the PG reserved bit. free page is invoked for each page that is freed to give them to buddy. The page count is set to one and free page is called to fake the buddy into thinking that this was actual freed by some kernel subsystem.

The memory allocator is general and doesn't know who is asking for the memory. Process context or Interrupt.

API offered by the buddy system for kernel internal allocation (kernel modules etc.). Get zeroed page, zeros the content and return the virtual address.

Get free pages increases the likelihood of fragmentation.

free page puts back a page into the free list. Failing to pass addr or order might spoil the kernel.

Allocations accept flags. GFP atomic won't put to sleep the asker, used for interrupt context. USER can put to sleep used for userspace. KERNEL is for kernel while BUFFER for buffers to pass data structures to userspace for example. Sleeping the kernel means the thread asking.

In case of NUMA we must also tell which numa node we want to allocate memory etc. or in case of Get free pages returns any numa node etc. Various NUMA policies are specified on how the node is chosen and can be set through set mempolicy. nodemask tell which nodes are involved and maxnode how many bits of the maks need to be checked. The modes can be default which asks for memory from the same node in which the request is issued. bind is from where I was spawned. Interleave different node on each allocation. Preferred you choose.

mbind sets a numa policy for a range of addresses.

move pages is used to move from one numa node to the other. A set of pages moved to a set of nodes and the status tells whether the call succeeded. The cache controllers of the various cores read and communicate for moving pages. 1 line is 64 Bytes therefore there is great overhead to move pages.

Issues are fragmentations (internal and external). Latency. For each numa node there is a buddy system. Anytime asking pages and releasing a spinlock is used.

Fast allocators allow quick allocation through caching. SLAB a block is used. SLUB is the default and is an optimisation of SLAB.

References

[a20()] A20 line. URL https://wiki.osdev.org/A20_Line.

[car()] Writing a bootloader from scratch. URL https://www.cs.cmu.edu/~410-s07/p4/p4-boot.pdf.

[con()] Context switching. URL https://wiki.osdev.org/Context_Switching.

[des(a)] Descriptor cache, a. URL https://wiki.osdev.org/Descriptor_Cache.

[des(b)] Interrupt descriptor table, b. URL https://wiki.osdev.org/Interrupt_Descriptor_Table.

[osd()] "8042" ps/2 controller. URL https://wiki.osdev.org/"8042"_PS/2_Controller.

[rma()] The workings of: x86-16/32 realmode addressing. URL https://web.archive.org/web/20130609073242/http://www.osdever.net/tutorials/rm_addressing.php?the_id=50.

[ser()] Interrupt service routines. URL https://wiki.osdev.org/Interrupt_Service_Routines.

[tas()] Task state segment. URL https://wiki.osdev.org/Task_State_Segment.

[vec()] Interrupt vector table. URL https://wiki.osdev.org/Interrupt_Vector_Table.

[wra()] Who needs the address wraparound, anyway? URL http://www.os2museum.com/wp/who-needs-the-address-wraparound-anyway/.

[x86()] Why doesn't linux use the hardware context switch via the tss? URL https://stackoverflow.com/questions/2711044/why-doesnt-linux-use-the-hardware-context-switch-via-the-tss.

[wik(2017)] Task state segment, Jun 2017. URL https://en.wikipedia.org/wiki/Task_state_segment.

[wik(2018a)] x86 memory segmentation, Mar 2018a. URL https://en.wikipedia.org/wiki/X86_memory_segmentation#cite_note-Arch-1.

[wik(2018b)] A20 line, Feb 2018b. URL https://en.wikipedia.org/wiki/A20_line.

[Bovet and Cesati(2006)] Daniel P. Bovet and Marco Cesati. *Understanding the Linux kernel*. OReilly, 2006.

[Brouwer()] Andries E. Brouwer. A20 - a pain from the past. URL https://www.win.tue.nl/~aeb/linux/kbd/A20.html.

[Collins(a)] Robert Collins. A20 - reset anomalies, a. URL http://www.rcollins.org/Productivity/A20Reset.html.

[Collins(b)] Robert Collins. The segment descriptor cache, b. URL http://www.rcollins.org/ddj/Aug98/Aug98.html.

[Duarte(2008a)] Gustavo Duarte. Memory translation and segmentation, Aug 2008a. URL https://manybutfinite.com/post/memory-translation-and-segmentation/.

[Duarte(2008b)] Gustavo Duarte. Cpu rings, privilege, and protection, Nov 2008b. URL https://manybutfinite.com/post/cpu-rings-privilege-and-protection/.

[Intel()] Intel. *Intel 64 and IA-32 Architecture Software Developer's Manual*, volume 3A.

[Oostenrijk(2016)] Alexander van Oostenrijk. Writing your own toy operating system: Jumping to protected mode, Sep 2016. URL http://www.independent-software.com/writing-your-own-toy-operating-system-jumping-to-protected-mode/.