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

Lecture 6: March 20

Lecturer: Alessandro Pellegrini *Scribe: Anxhelo Xhebraj*

The data structure pglist_data holds the informations about all the available memory of a Numa Node.

For each node there are different nodes. Within a physical numa node we have node_zones that describes. Then there is the map and the size of the node.

First zone is DMA, on x86 32 bit it is associated with the first physical 16MB. This is for Direct Memory Access. This is done because some DMAC are not capable of accessing other regions of memory. Linux reserves this for example also for disk access. Zone Normal is all the memory that the kernel has always mapped to its own virtual memory. Zone HIGHMEM is sometimes mapped by the kernel on some cases. It is usually used to remap. In x64 the notion of high memory is removed since the availability of addresses makes it easy to access portions of physical memory that are not mapped.

Zones are initialised after the kernel page table is setup by paging _ init. We need a way to describe the physical available memory. The kernel has still some init areas that need to be wiped etc. The kernel must know which free pages are available. The zones are a description of available physical memory.

In the node version of the api is used for initialising the data structure of a node. Each node has the data structure stored in its own.

During the POST phase the BIOS can know how much physical memory is available and setups a table called e820 table, stored in memory that has info related on how much physical memory is available and which are the regions available or not (for example in BIOS shadow initialisation telling ”there is the bios here”). PFN is the number of physical available memory in ram. The 3rd entry is used by the kernel and the very beginning by the IVT.

The description of a node has a zone_t data structure that has free area entry inside, that is used by allocators. It has also a spinlock that might be a bottlenech.

For each node there is a pg data t which has zones (usually 3). Each zone has a memory map that tells which are the available pages in that zone.

Core Map is used to describe the memory in ZONE NORMAL, tells how much available physical memory in ZONE NORMAL there is. You have to look this Data Structure to know how to allocate memory. We're talking about kernel dynamic memory allocation. There is no heap in the kernel.

It has a list that has the list of pages, usage count and some flags. The list is a free list. All the available physical frames are kept in a free list making the allocation quick not having to check some bitmap. The counter counts the virtual references mapped to a frame. One single frame could be mapped by multiple virtual addresses for example in memory sharing. The flags describe how that specific page can be used. (list head keeps the pointer to next and previous).

Free area initialisation is done in free area init. The frame has no virtual reference and the frame is reserved. Since both bootmem and the steady state allocators are both working we must be careful to not have conflicting actions between the two. By PG reserved we ensure that the stedy state allocator will not do anything until bootmem is ”removed”. Mem init will then reset that flag.

6.1 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.

6.2 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/`.