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

Lecture 7: March 23

Lecturer: Alessandro Pellegrini *Scribe: Anxhelo Xhebraj*

As described, even though the Buddy Allocator is a fast algorithm to manage memory for the kernel it still has some problems that aren't really addressed:

- **Internal Fragmentation:** produced by the granularity at which the Buddy System works (page size). If the kernel needs just a buffer of size less than that of a page, it still has to allocate the page fully, meaning that the remaining space cannot be used by other allocations.
- **External Fragmentation:** after many allocations of small size it might happen that the Buddy System is not able to coalesce some buddies into bigger blocks of contiguous memory meaning that if at some time the kernel needs a large block of contiguous memory, such request cannot be fulfilled even if there are a set of *not contiguous* pages that together make the size of the request.
- **Spinlock Contention:** the allocation in the same memory zone within a node is serialized since each allocation needs to acquire `zone_t->lock`. This can become a bottleneck if multiple kernel threads are trying to allocate memory.

7.1 Quicklists

Quicklists consist in pre-allocating pages for each CPU. The list of free pages is kept in a *per-cpu variable* meaning that each CPU will have its own variable with its own list of pages. This new technique of allocating pages increases the parallelism of the allocation with respect to the Buddy System since threads running on different CPUs do not have to dispute on the zone lock but can take pages from their own free list. It was mainly used for page table pages allocations such as `pgd_alloc` etc.

The allocation of a page through quicklists is done through the `quicklist_alloc` function.

```
> include/linux/quicklist.h
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;
    }
```

In the allocation `get_cpu_var` is called to get the reference to the free list assigned to the CPU the thread is currently running on. In doing so preemption is disabled in `get_cpu_var` in order to not allow the thread to be moved on another CPU with a reference to a variable which is specific to the CPU it is currently running on, otherwise concurrency problem might arise.

`q->page` is the first page of the free list. It is checked whether the free list is empty (`NULL`) through `if(likely(p))`. The `likely` macro informs the compiler that the flow of execution will take the branch path meaning that *likely* the code block inside the `if` will be executed therefore the compiler should take care to optimize the branch prediction. Inside the `if` code block the *head* (first element) of the list is taken and the pointer of the list is set to the pointer to the next page held inside `p[0]` and the number of free pages is decremented (`q->nr_pages--`).

Finally `put_cpu_var` will re-enable preemption. If there was indeed a page available in the free list then it is returned, otherwise the allocation falls back to the Buddy System through `__get__free_page()`.

Quicklists were used in the i386 architecture from kernel v. 2.6.22 up to 2.6.24.7 and have then been replaced by per-cpu pages/hot-n-cold pages.

7.2 SLAB Allocator

To make more efficient the allocation of small frequently allocated/deallocated data structures in the kernel such as `pid, mm_struct` the *slab* allocator (and in later versions also the *slob* and *slub*) was introduced.

The main goals of this new allocator are:

- Amortize the internal fragmentation produced by the Buddy System allocator for small allocations.
- Improve access performance to kernel object by aligning them to L1/L2 caches.
- Cache commonly used objects to alleviate the system from allocating, initializing and destroying objects.

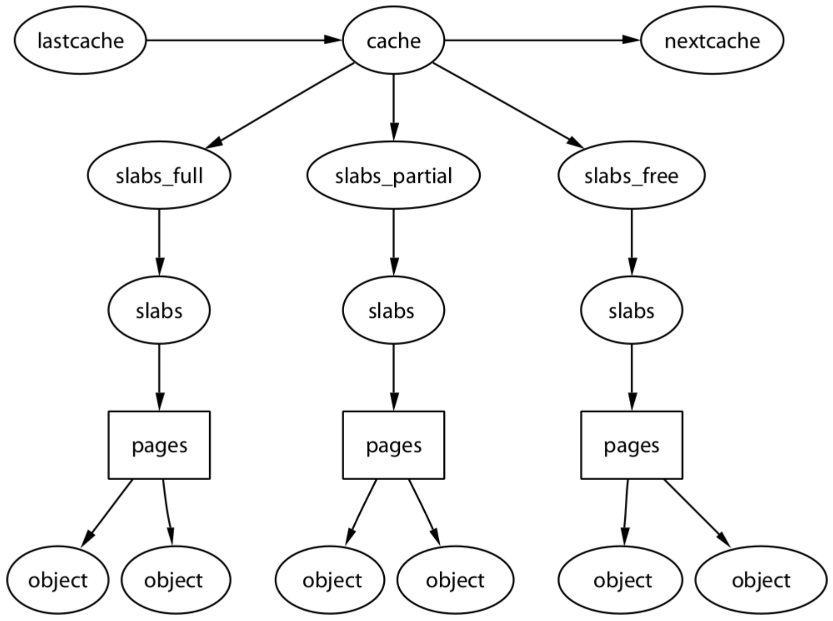

Figure 7.1: Slab structs relationships. ([?] pp. 16)

There is a list of caches where each cache keeps a slab of a specific size. A cache is organised in three types of slabs: full (Don't have buffers to be given), partial (some used and some free), free (deallocated or never allocated). Objects are abstractions over pages.

7.2.1 SLAB Interfaces

Found in linux / malloc h. kmalloc asks for a given size and returns the virtual address of a buffer of that size. kfree frees memory allocated via kmalloc. k malloc node is an api to the slab allocator that in the end will ask to the buddy system of a specific numa node for allocating some page.

kmalloc should be used for frequent allocations and deallocations of the same size.

up to kernel v 3.9.11 there was the struct cache size. cs_cachep is a pointer to the memory. There is a table of multiple size fixed-size caches.

In kernel 3.10 we move from fixed size to list with spinlock again. You can either have a shared across cores allocator or one allocator for each core. Having one allocator for each core requires space. What's the difference btw using the buddy system and slab? buddy has one spinlock for each numa node while slab has one spinlock for each size of cache.

SLAB and Buddy both are for the kernel

Per node cache coloring: size of object then padding and so on. Why is coloring used? to align objects to L1 Cache Bytes. Two objects of the same size will not fall in the same cache line. An object of size greater than one line is padded to the size of multiple cache lines.

This ensures that two slabs objects allocated will not fall in the same cache line to not fall into cache contention.

Members that are accessed together and used frequently together (Common Members) are placed close together to optimise cache hits for example the spinlock and slab partial in kmem cache node struct.

(Loosely related fields): due to the false cache sharing problem we have that cache controllers in order to be coherent tell the others cache controllers that they are going to write that line wanting "mutual exclusion". With coloring we ensure that different buffers will not fall in the same cache line.

7.2.2 Cache flush operations

Similarily to the TLB relies on the hardware specific operations for granularity and coherency of the flushing. There are also problems because the hardware cache uses virtual addresses, therefore two processes addressing the same virtual address might have a cache hit to a region of memory that is not the physical one they wanted to access. After flushing the page cache we must also flush the TLB.

`flush_cache_all`: Flushes the entire CPU cache system. It is used for when global data structures, for example kernel page tables, are changed to ensure cache coherency.

others...

What is the best way to devise a cache? physical or virtual? Intel architectures use Virtual addresses to tag L1 cache. If there is a miss in L1 the TLB is consulted to get the physical and check the L2 which is addressed through the physical address. There is a protocol btw L1 and TLB to know whether a virtual address is consistent with the current paging scheme. Therefore in intel we do not care about cache consistency.

Virtual aliasing is the problem described above where we tag the L1 cache through virtual addresses but if the cache is not coherent ...

The other apis are a low level api that are used by the description above.

copy from and to user ensures that the copy of memory is done correctly since there might be a process switch and the write might be writing memory of another process.

Access ok checks whether the memory area passed is correctly mapped to that process.

`vmalloc` used to map some memory in the kernel in a stable way, that will be used for a long time. No idea about the memory contiguousness. No info about the organisation of physical frames. Used for usually loading some kernel module for code, data etc of the module. It doesn't rely in either the Buddy system or slab.

`virt_to_phys` and viceversa used in kmalloc or get free page to compute the mapping btw physical and virtual addresses. This is done to be hardware independent when developing a kernel module to not rely on offsets etc.

For allocation size in kmalloc is limited to 8KB in Linux. vmalloc btw 64/128MB. Kmalloc is physical contiguous while vmalloc no. Vmalloc invalidates transparently the TLB etc while kmalloc no. This is done because for example in loading kernel modules you want that all threads have visibility of the change.

After setting up all the memory `trap_init()` initialises the IDT and GDT. The only entry point to kernel land from user space is through the interrupt `0x80`. The same is done in Windows.

That interrupt executes the system call dispatcher which finds out what the user code wants to do. Every system call has a code assigned to it.

Interrupts automatically reset the Flags while Traps do not. The dispatcher explicitly clears interrupts through `cli` instruction. In multi-core systems this is not enough, we must ensure correctness of all the data structures of the kernel. Spinlocks (implemented through the `cmpxchg` instruction) are used to access data structures.

Since syscalls have preassigned numbers if we change only one of them we break backward compatibility.

Macros are used for generating asm volatile blocks defining a syscall. Multiple macros depending by the number of the parameters of the syscall (usually at most 6 parameters).