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 From the kernel version 2.6.16 the `pgdat_list` has been replaced by a global array called `node_data[]` and the iteration over it is done through macros defined in `include/linux/mm/zone.h`.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.

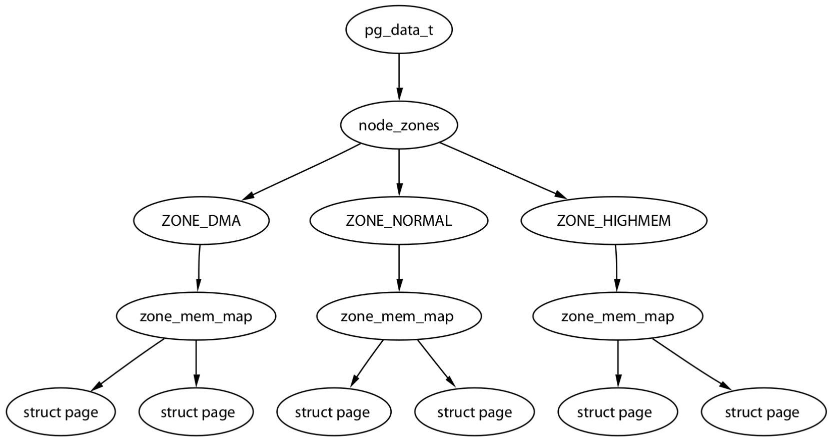

Figure 6.1: Relationships between structs. Note that there are multiple `pg_data_t` linked together. ([Gorman(2004)] pp. 16)

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`.

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

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*`) will eventually call `free_area_init_core()` (`mm/page_alloc.c`) that performs the setup of the data structures described below.

`free_area_init*` is indirectly called by `paging_init`

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 and is used by the buddy system to protect the data structures from concurrent access.

`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

([Bovet and Cesati(2006)] pp. 302), ([Mauerer(2010)] pp. 14, Sec. 3.5.1) ([Gorman(2004)] Chap. 6)

The kernel subsystem that handles the memory allocation requests for groups of contiguous page frames is called the *zoned page frame allocator*.

The component named "zoned allocator" receives the requests for allocation and deallocation of dynamic memory. In the case of allocation requests, the component searches a memory zone that includes a group of contiguous page frames that can satisfy the request. Inside each zone, page frames are handled by a component named "buddy system". The buddy system of all zones and nodes are linked together via the allocation fallback list (`node_zonelists`). When a request for memory cannot be satisfied in the preferred zone or node, first another zone in the same node, and then another node is picked to fulfill the request.

Free memory blocks in the system are always grouped as two buddies. The buddies can be allocated independently of each other; if, however, both remain unused at the same time, the kernel merges them into a larger pair that serves as a buddy on the next level.

The buddy system uses the `free_area[]` field of `zone_t` to perform memory allocation and deallocation. `free_area[]` is an array of structs defined as follows.

```
> include/linux/mmzone.h (v. < 2.6.10)
typedef struct free_area_struct {
    struct list_head    free_list;
    unsigned long   *map;
} free_area_t;
```

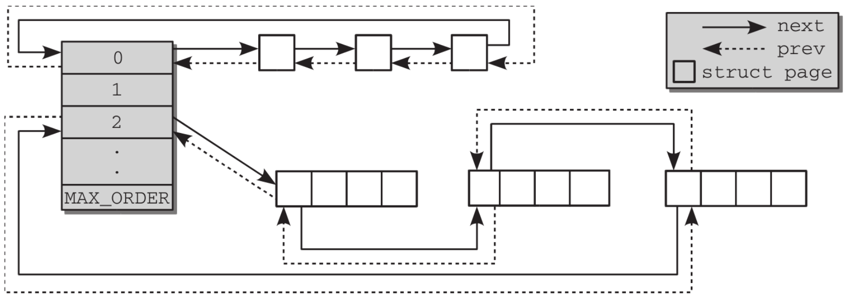

Figure 6.2: `free_area[]` array data structure representation within a zone

Each entry of the array is associated to an *order* and holds in a list pointed by `free_list` memory blocks of size 2^{order} composed of contiguous **free** `struct pages`. Therefore order 0 contains blocks of single pages, order 1 contains blocks of 2^1 pages, order 2 blocks of 2^2 pages, up to `MAX_ORDER - 1` (`MAX_ORDER` usually set to 11). The list is formed by linking the blocks of the same size together through the `list` field of the first `struct page` belonging to the block.

The `map` field of the struct points to a bitmap used for allocation and deallocation. Each bit represents a pair of buddies in the current order. It is set to 0 if both are allocated or both are free and it is set to 1 otherwise.

Allocation

When a request for allocating an area of a given order `ord` is issued, the buddy system tries to find a free block inside of `free_area[ord].free_list`. If the request can be fulfilled in the requested order `ord` then the block is returned and the bit in the bitmap is toggled. Otherwise the buddy system recursively searches for free blocks in higher orders, say it is found in `high_ord > ord`, and splits the block into two blocks (called buddies), one put in the list of `high_ord - 1` (also toggling the bits in the bitmap) and the other recursively split into smaller blocks (buddies) until `ord` is reached.

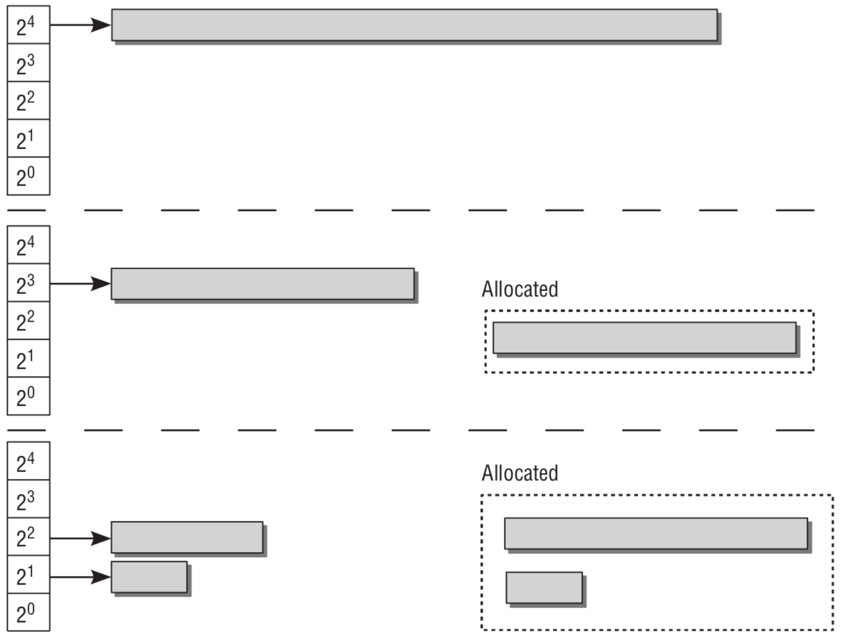

Figure 6.3: Evolution of `free_area[]` when first requesting a block of order 3 and then of order 1.

Deallocation

When the block is later freed, the buddy will be checked. If both are free, the kernel will try to *coalesce* the buddies together immediately. In the worst scenario the operation requires `MAX_ORDER` steps. To detect if the buddies can be merged or not, the bit corresponding to the affected pair of buddies in the bitmap is checked. As one buddy has just been freed by this function it is known that at least one buddy is free. If the bit in the map is 0 after toggling, we know that the other buddy must also be free and thus they can be merged. Otherwise if the bit is 1, the other buddy is in use therefore the buddies cannot be merged and the block goes into its order list.

All the memory operations performed by the buddy system require the acquirement of the `lock` found in `zone_t` therefore it can be a bottleneck for the responsiveness of the system since there can be contentions on the spinlock.

Even though the algorithm is very fast, the frequent allocation and release of page frames and blocks may lead to a situation in which several page frames are free in the system but they are scattered (not contiguous) throughout the physical address space. Single reserved pages that sit in the middle of an otherwise large continuous free range can eliminate coalescing of this range very effectively.

The finalization of memory management initialization is done in `mem_init()`.

```
> arch/i386/mm/init.c
mem_init()
    ...
    free_pages_init()
        ...
        > mm/bootmem.c
        free_all_bootmem()
            free_all_bootmem_core() {
                ...
                for (i = 0; i < idx; i++, page++) {
                    if (!test_bit(i, bdata->node_bootmem_map)) {
                        count++;
                        ClearPageReserved(page);
                        set_page_count(page, 1);
                        __free_page(page); //decrements page count and if 0
                                           //adds page into the free list
                    }
                }
                ... // free also bootmem map
            }
```

For each unallocated page in bootmem, the page flag `PG_reserved` is cleared (it was previously set on initialization by `free_area_init()` as said previously) and the `count` is set to 1 in order to fake `__free_page()` into thinking that this is an ordinary deallocation for the buddy system which on decrementing `count` and checking that is 0 will put it into the free list.

6.2.1 Zoned Allocator APIs

([Gorman(2004)] Chap. 6), ([Mauerer(2010)] Sec 3.5.4)

```
> mm/page_alloc.c
// Allocation
alloc_pages(gfp_mask, order)
    Used to request 2^order contiguous page frames. It returns struct
    page * of the first page of the block

get_zeroed_page(gfp_mask)
    it expands to alloc_pages(gfp_mas | __GFP_ZERO, 0) but instead of
    returning a struct page * it returns the linear address of the page
    frame allocated. The contents of the page are set to 0

__get_free_page(gfp_mask)
    Allocates one page frame and returns its linear addres

__get_free_pages(gfp_mask, order)
    similar to alloc_pages(gfp_mask, order) but returns the linear
    address of the first page frame of the block
```

```
// Deallocation
free_page(addr)
    frees a page given its linear address

free_pages(addr, order)
    frees a block given its linear address and its order

__free_pages(page, order)
    given a struct page * if its PG_reserved is set to 0 (page not
    reserved) decreases the count field and if it becomes 0 it assumes
    that 2^order contiguous page frames starting from the one corresponding
    to page are no longer used

__free_page(page)
    expands to __free_pages(page, 0)
```

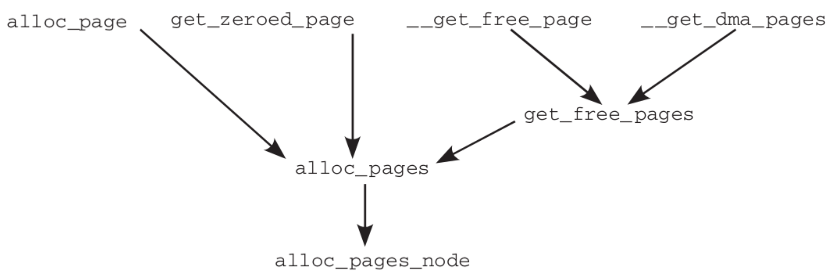

Figure 6.4: Relationships between allocating functions

[prolinkern pp. 220 for flags]

The `gfp_mask` parameter is a mask consisting of a set of flags that specifies to the zoned allocator the allocation context, such as *zone modifiers* of the allocation (`__GFP_DMA`, `__GFP_HIGHMEM`) or others such as

```
GFP_ATOMIC
the allocation cannot be interrupted (the process cannot be put into sleep)
since this is a urgent allocation.

GFP_KERNEL and GFP_USER
respectively used to allocate kernel memory and userspace memory

GFP_NOFS (previously GFP_BUFFER)
generic page, usually used to allocate a buffer
```

6.3 High Memory

High memory allocations can be performed only through the subset of APIs described above which return `struct page *`. This is done because page frames in the `ZONE_HIGH` area (above 896MB) do not have a permanent mapping in the linear address space of the kernel.

The kernel uses three different mechanisms to map page frames in high memory; they are called *permanent kernel mapping, temporary kernel mapping* and *noncontiguous memory allocation* (this one described later). Memory allocation for high memory is performed through `alloc_pages()` and `alloc_page()` APIs described above. These APIs return the linear address of the page descriptor (`struct page *`) of the first allocated page frame which always exist because all page descriptors are allocated in *low memory* (`0:896MB`). The kernel then uses part of the last 128MB of its linear address space to map high-memory page frames. Trivially this kind of mapping is temporary otherwise only 128MB of high memory would be accessible.

([Mauerer(2010)] pp. 250) `vmap(struct page **pages, int count, unsigned long flags, pgprot_t prot)`

Given an array of `struct page *` (even not contiguous) creates the entries in the page tables to access the pages through a virtually contiguous space. When freeing `vmap`ped area through `vunmap` eventually `vmfree_area_pages` (`mm/vmalloc.c`) will be called which will also flush the TLB on all cores.

([Mauerer(2010)] Sec. 3.5.8) `kmap(struct page *)`

Given a page descriptor, maps the physical addresses of that page to the virtual address space of the kernel in highmem starting with virtual address `PKMAP_BASE`. These pages are also known as "Permanent Kernel Mappings". There is a dedicated Page Table (last level) in the kernel page tables of which address is stored in `pkmap_page_table` to handle this kinds of mappings.

An array of counters `pkmap_count[LAST_PKMAP]` each referring to an entry of `pkmap_page_table` is used for management. If a counter is 0 it means that the corresponding PTE is not used for mapping any high-memory page frame and is usable. If it is 1 it means that the PTE does not map any high-memory page frame but it cannot be used since the corresponding TLB entry has not been flushed since its last usage. If the counter is n the corresponding PTE maps a high-memory page frame which is used by exactly $n-1$ kernel components ($n-1$ components have called `kmap` on the same `struct page *`. Note that if a page frame was already `kmap`ped and `kmap` is called again on the very same page frame, then the second call will return the same virtual address of the first).

When mapping a page frame with no corresponding virtual address the function tries to find an empty PTE in `pkmap_page_table` (i.e. a PTE with counter 0), creates the entry, sets its counter to 2, and returns the virtual address assigned. When unmapping through `kunmap` the counter is decremented (therefore it will be 1 when no kernel thread references that page anymore). The TLB is flushed on all cores through `flush_tlb_all()` only when all the PTEs have counter 1. No errors caused by TLB caching can arise since if over time two threads perform `kmap`$_1$ `->` `kunmap`$_1$ `->` `kmap`$_2$ (where the subscript tells the id of the thread performing the call and arrows indicate the temporal sequence of events) with in betweeen them any kind of operations, and both `kmap`$_1$ and `kmap`$_2$ return the same virtual address there must have been a `flush_tlb_all()` in between the first unmapping and the second mapping according to the algorithm.

Note that in case of not available PTEs the kernel thread issuing a `kmap` might be put to sleep until one is available (`kunmap` issued). Also since the number of available PTEs are those fitting in one page frame the function must be used for mappings of not too prolonged time.

`kmap_atomic(struct page *page, enum km_type type)`

The `kmap` function described above must not be used in interrupt handlers because it can lead to sleep. The kernel therefore provides this alternative mapping function that executes atomically. When a thread calls this function it becomes not preemptable until it issues `kunmap_atomic`. To make a thread not preemptable the function increases its `preempt_count` field. Before returning the function ensures that the TLB of the processor on which the thread runs doesn't have cached anything on the virtual address it is going to return by calling `__flush_tlb_single(vaddr)`. By this fact, the mapping is not visible to other processors. The function uses a portion of virtual memory called fixmap.

6.4 NUMA Allocation Policies

Starting from kernel v. 2.6.18, there were added system calls that allowed userspace to specify a policy to honor during allocations. Note that such policies are followed also by the functions described above. The implementation of this system calls can be found in `mm/mempolicy.c` through the macro `SYSCALL_DEFINE*` which will be explained in future lectures. By policy it is meant on which node the kernel should allocate memory in a NUMA system. One of the main functions is `set_mempolicy(int mode, unsigned long *nodemask, unsigned long maxnode)` which sets the memory policy of the calling thread. The `mode` parameter can be one between

`MPOL_DEFAULT`

Sets the policy for the calling thread to the system's default policy which tries to allocate memory from the node requesting it and then to close nodes. When specified in the `mbind` function (see below) it follows the thread policy.

`MPOL_BIND`

This mode specifies that memory must come from the set of nodes specified by the policy through `nodemask` and `maxnode`. Memory will be allocated from the node in the set with sufficient free memory that is closest to the node where the allocation takes place.

`MPOL_INTERLEAVED`

This mode specifies that page allocations be interleaved, on a page granularity, across the nodes specified in the policy.

`MPOL_PREFERRED`

This mode specifies that the allocation should be attempted from the single node specified in the policy. If that allocation fails, the kernel will search other nodes, in order of increasing distance from the preferred node based on information provided by the platform firmware.

`mbind(void* addr, unsigned long len, int mode, unsigned long *nodemask, unsigned long maxnode, unsigned flags)` sets a NUMA policy for a range of addresses starting from `addr` to `addr + len`.

The system call `move_pages(pid_t pid, unsigned long nr_pages, void **pages, const int *nodes, int *status, int flags)` is defined in `mm/migrate.c` to allow to move pages from one NUMA node to another. Note that this is an expensive operation that requires the cache controllers of the various cores to perform the migration of pages working at 64Bytes per time. For more information about NUMA policies refer to `Documentation/vm/numa_memory_policy.txt` and manpages.

6.5 Conclusions

In this lecture various parts of the memory management of the linux kernel were explored. While the code seen in the previous lectures were about specific, older versions of the kernel, the concept shown in this lecture apply also to modern versions of it. One main observation has to be done: as anticipated the concept of high memory thankfully is not present in 64 bit systems since the virtual address space of the kernel is more than enough to address all the possible physical memory.

The concept of NUMA system is introduced in the kernel which brings in new problems and interfaces in the system programming world such as the concept of NUMA policies. In linux UMA systems are treated as NUMA systems having just one node. In this context allocations are driven by NUMA policies combined with the Buddy Sytems.

When an allocation needs to be performed, the current NUMA policy for the thread issuing the request is queried combined with the eventual zone modifier defined by the `gfp_mask` of the request to choose the Buddy System that has to fulfill the request. There is one Buddy System per zone within a node and the requests to one buddy are serialized, i.e. managed through the lock in `zone_t`.

References

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

[Gorman(2004)] Mel Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[Mauerer(2010)] Wolfgang Mauerer. *Professional Linux Kernel Architecture*. Wiley, 2010.