Linux Memory Management

Luca Abeni

luca.abeni@santannapisa.it

Memory Management in the Kernel

- In user space, we are used to `malloc()`, `new` and friends
  - What we see is virtual memory
  - Easy to allocate arbitrary amounts of memory
  - Lazy memory allocation and advanced features, ...
- The OS kernel is the one generally implementing virtual memory
  - For the sake of simplicity, let's forget μ-kernels and hypervisors
- How is virtual memory implemented?

Physical Memory and Virtual Memory

- The kernel directly accesses the hardware
  - It manages physical memory
- The kernel provides functionalities to user-space
  - It manages virtual memory too
  - It handles the translation of virtual addresses into physical addresses
    - MMU configuration, page faults handling, etc...
- So, the kernel contains both a virtual memory and a physical memory manager!

Paging

- Translation of virtual addresses into physical addresses is generally performed using *paging*
  - The MMU uses a *page table* for the translation
    - Can be a complex data structure (hierarchical paging)
  - The kernel is responsible for managing the page table
- Physical memory allocator: allocates physical pages of memory
- Virtual memory allocator: allocates virtual memory ranges

Memory Allocator

- Goal: allow to allocate memory buffers of specified size
- Simplest idea: list of free memory fragments
  - Ordered by size: makes allocation easier
  - Ordered by memory address: makes deallocation (compacting adiacent fragments) easier
- In general, a single list of free memory fragments is not a good idea...
- Better idea: multiple lists (for different fragment sizes)

Multiple Free Memory Lists: Buddies

- Constraints: memory fragments have sizes power of 2
- Multiple lists, containing fragments with different sizes
- The i^{th} queue contains fragments of size 2^{b+i}
- Allocation of buffer of size s:
  - Find the smallest i such that $2^{b+i} > s$
  - If the i^{th} queue is not empty, return a memory fragment from it
  - Otherwise, split a fragment from the $(i+1)^{th}$ queue, and insert 2 fragments in the i^{th} queue. Then allocate one of them
    - Might split a fragment from the $(i+1)^{th}$ queue if needed (and so on)

Buddy Allocator: Deallocation

- When a fragment from the $(i+1)^{th}$ queue is split in 2 fragments of the i^{th} queue, such fragments are named *buddies*
- Generally, when a fragment is split one of the two buddies is used
  - When it is released, the two buddies can be recompacted
- On free, it is easy to see if the buddy of the freed fragment is in a list
  - Need to compute the buddy address...

Buddy Allocator and Pages

- The i^{th} list contains fragments of 2^i pages
  - i: order of the allocation
- At the beginning, only the highest-order list (say, list m) is not empty
- When a i-order allocation is requested, a fragment from list m is split in two buddies
  - One is inserted in list $m-1$, the other one is split in 2 buddies...
  - ...And so on, until buddies are inserted in list i.
  - Then, a memory fragment composed by 2^i pages is allocated (and the other one remains in the i^{th} list

Buddy and Pages: Deallocation/Merging

- When a memory fragment is freed, need to check if its buddy is free too
  - In this case, they can be merged!
- Order i deallocation: the fragment is composed by 2^i pages...
  - Look at the page number of the first page of the freed segment: the i rightmost bits are 0
  - Then look at bit i: the buddy will have this bit swapped
  - So, `buddy_number = page_number ^ (1 << i)`
- The merged fragment has order $i + 1$ (so, it has the rightmost $i + 1$ bits set to 0)
  - `merged_number = page_number & buddy_number`

Physical Memory Allocator in Linux

- Allocates fragments composed by contiguous physical pages
  - A physical page is sometimes known as *page frame*
- It is not possible to allocate arbitrary amounts of memory
  - Only fragments composed by 2^i pages
  - i is the *allocation order*
  - Special case: allocate 1 physical memory page (0-order allocation)
- Linux uses a buddy allocator for physical pages

Allocating Physical Pages

- 2^i pages can be allocated with

  ```
  struct page *alloc_pages(gfp_t m, unsigned int i)
  ```

  - `i` is the order of the allocation
  - `m` indicates which kind of pages to allocate, and how
- The return value is a pointer to a `struct page`, describing the first physical page of the fragment
  - Each physical page is described by a `page` structure, also identified by a *page frame number* (pfn)
  - There are functions to convert a pointer to `frame` structure into its pfn, and vice-versa
  - The conversion depends on the *memory model*

Allocating Physical Pages — 2

- `alloc_pages()` returns the pointer to a `struct page`
- What to do to actually access the content of the page?
  - We need to know the virtual address where the page is mapped...
  - Can be computed with

    ```
    void * page_address(struct page *page)
    ```

- `__get_free_pages()` combines `alloc_pages()` and `page_address()` ...
- ...Casting the result (a pointer to `void`) to `unsigned long`

Allocating One Single Physical Page

- Two functions specialized for 0-order allocations:
  - `struct page *alloc_page(gfp_t gfp_mask)`
  - `unsigned long __get_free_page(gfp_t gfp_mask)`
- They end up invoking `alloc_pages()` and `__get_free_pages()` with second parameter equal to 0

Memory Zones

- Linux organizes the physical memory pages in *zones*
  - Zone: set of pages with similar properties
  - Which properties? Can be used by DMA devices, can lack a mapping to virtual pages, ...
- `DMA` and `DMA32` zones: the pages can be accessed by DMA/bus mastering devices
- `HIGHMEM` zone: the pages are not always mapped in the virtual address space
  - What? A physical page not mapped in a virtual page??? 32bit systems (4GB virtual address space) with more than 4GB of RAM
  - Possible on 32bit x86 CPUs by Intel, thanks to something called “PAE”

Get Free Pages Flags

- All the allocation functions have an argument of type `gfp_t`: the gfp mask
  - gfp stands for get free pages
- This is a bitmask that can contain multiple flags
- Some flags specify where to allocate the memory from
  - `__GFP_DMA`, `__GFP_DMA32`, `__GFP_HIGHMEM`
- Some other flags specify constraints for the allocator
  - `__GFP_WAIT`, `__GFP_IO`, `__GFP_NOFAIL`, ...
- Some constants combine important gfp flags:
  - `GFP_ATOMIC`, `GFP_NOWAIT`, `GFP_NOIO`, ... `GFP_KERNEL`, `GFP_USER`, ...

Virtual Memory Allocator in Linux

- `kmalloc()/kfree()` and `vmalloc()/vfree()` allow to allocate *arbitrary amounts* of memory in the virtual address space
  - Difference: `kmalloc()` allocates contiguous physical memory, while `vmalloc()` allocate fragments of virtual memory that might be non-contiguous in physical memory
- They are based on `get_free_pages()/get_free_page()` at the lower level
- Upper layer to support allocation of memory fragments with size different from 2^i pages

Details on kmalloc()

- If the size of the memory to be allocated is larger than a `KMALLOC_MAX_CACHE_SIZE`, then round it up to 2^i pages and call `get_free_pages()`
  - See check in `include/linux/slab.h::kmalloc()`
  - Otherwise, allocate memory from a *cache of allocated objects* (slab)
- In any case, the allocated memory is contiguous in both physical and virtual memory!
  - A “linear mapping” can be used to convert between virtual and physical addresses
  - No need to modify the page table...

Details on vmalloc()

- Physical memory is allocated by invoking `get_free_page()` multiple times
  - So, it is not necessarily contiguous in physical memory!
  - No “linear mapping”; need to modify the page table to make the memory region contiguous in virtual memory
- Higher overhead than `kmalloc()` (page table modifications), but easier to allocate large buffers
- Can use `kmalloc()` internally, for its own data structures

Caching Memory Allocations

- The kernel often allocates/deallocates similar objects a lot of times
  - Think about `skbuf`s, `task_struct`s, `inode` structures, `dentry` structures, ...
- To avoid the cost of fully allocating/initializing them all the times, some caching mechanism can be used
  - Cache of allocated physical pages (when freed, cache them instead of returning them to the buddy allocator)
  - Cache of deallocated "memory objects"

Slabs

- The buddy allocator can only allocate 2^i pages (i: order of the allocation)
- How to allocate arbitrary amounts of memory?
  - Need for an additional software layer over the buddy allocator
  - Allow to allocate “memory objects” of various sizes
  - Support different object sizes
- slab: portion of memory containing multiple memory objects, all of the same size
  - slab size: multiple of the page size, depending on architecture and allocator

Slabs and SLAB

- Software layer handling slabs
  - Allocating/caching objects
  - Requesting physical pages to the buddy allocator
- Originally called SLAB
  - So, there is a SLAB allocator working on slabs...
  - But SLAB != slab...
  - ...Confusing!
- Now, SLUB and SLOB are also available
  - So, there are 3 different slab allocators: SLAB, SLUB and SLOB!!!
  - What a mess...

SLAB, SLUB and SLOB

- SLAB, SLUB, and SLOB are all *slab allocators*
  - So, they all export the same API
  - What changes is the the internal implementation
- They differ in how slabs are internally managed, and how objects are cached
- To be precise, SLOB is not actually a slab allocator: it exports the API of a slab allocator, but does not internaly use slabs...

Objects, slabs and Caches

- slabs are stored in *caches*
- Cache: manager for allocating objects of a given type
  - All objects in a cache have the same size
- The main difference between SLUB and SLAB is in how the slab caches are organized (a single list vs multiple lists, ...)
- Try "`sudo cat /proc/slabinfo`" to have an idea of the caches present in your system
  - The "`kmalloc-*`" caches are used... By `kmalloc()`!!!

Allocator API

- `kmem_cache_create()`: creates a new object cache
- `kmem_cache_shrink()`: removes free slabs from a cache, freeing pages
- `kmem_cache_alloc()`: allocates an object from the cache
- `kmem_cache_free()`: frees an object returning it to the cache
- `kmem_cache_destroy()`: deallocates all the objects allocated from a cache, and destroys the cache
- `kmalloc()` and `kfree()` are based on these...
  - How to support arbitrary sizes? They use multiple caches... Will see later

The Linux SLAB Allocator

- Implements a slab allocator as a set of caches sharing no data
  - Per-cache locking
- Evey cache has 3 lists:
  - Full slabs list (slabs containing no free objects): `slab_full`
  - Partial slabs list (slabs containing some allocated objects and some free objects) :`slab_partial`
  - Free slabs list (slabs containing only free objects): `slab_free`
- The Linux kernel is NUMA aware: 3 slab lists per NUMA node!

The SLAB Cache

- The slab interface is described in `include/linux/slab.h`; the SLAB details are in `include/linux/slab_def.h` and `mm/slab.h`
- `struct kmem_cache` in `include/linux/slab_def.h`
  - Contains some cache arguments and the cache state
  - Also contains an array of `kmem_cache_node` structure (they contains the 3 lists!)
- slabs are enqueued in these lists
  - Actually, the first page of each slab is enqueued
  - See the `slab_list` field in `struct page`

Using the 3 Lists

- Objects are generally allocated from slabs in `slab_partial`
- If `slab_partial` is empty, slabs from `slab_free` can be used
  - After allocating the object, the slab is moved to `slab_partial`
- If `slab_free` is also empty, invoke `__alloc_pages()` (actually, `__alloc_pages_node()`) to allocate a slab
- When an object is freed, add it to its slab
  - If it was the last allocated object of the slab, move the slab to `slab_free`

Multi-Core Optimization

- The original SLAB algorithm was designed for uni-processor systems
  - Per-cache locks protecting the 3 lists (and other `kmem_cache` fields
  - On multi-core systems, scales badly (high risk of lock contention)
- Optimization: per-CPU (actually, per-core) cache of free objects
  - See the `cpu_cache` field of `kmem_cache`
  - Can be accessed without locking, but is “percpu” (disable preemption)

Example: Allocating an Object

- `kmem_cache_alloc()`, defined in `mm/slab.c` invokes `slab_alloc()`
- `slab_alloc()` invokes `__do_cache_alloc()` which invokes `____cache_alloc()`
- `____cache_alloc()` looks at the per-CPU cache (using `cpu_cache_get()`
  - If the per-CPU cache is not empty, returns a free object from it (`ac->entry[--ac->avail]`)
  - If the per-CPU cache is empty, refill it (`cache_alloc_refill()`)

Refilling the per-CPU Cache

- `cache_alloc_refill()` is invoked when the per-CPU cache is empty and an object has to be allocated
- It searches for a slab to be used (from some of the lists, or from the buddy allocator)
- Then, it invokes `alloc_block()` (to fill the `per_cpu` array with objects) and `fixup_slab_list()` (to insert the slab in `slabs_full` or `slabs_partial`)
  - `fixup_slab_list()` is eventually called by `cache_grow_end()`

Slabs and Coloring

- A slab contains multiple objects
  - The slab is some pages large
  - The slab size is generally not an integer multiple of an object size
  - So, the first object can have an offset respect to the beginning of the slab
- To be more hw-cache friendly, each slab has objects starting at a slightly different offset
  - Goal: distribute buffers evenly throughout the cache

Coloring Example

- When a slab is initialized, the first buffer starts at a different offset from the slab base (different color)
- This results in different colors because slabs are page-aligned...
- Example: 200-byte objects, with 8-bytes alignment requirement
  - Slab 1: objects at offsets 0, 200, 400, ...
  - Slab 2: objects at offsets 8, 208, 408, ...
  - Slab 3: objects at offsets 16, 216, 416, ...
- When the maximum offset is reached, restart from 0

SLUB

- SLUB allocator: born to simplify the SLAB code
  - The SLAB complexity went... Kind of out of control
- Avoid multiple queues: all the slabs are in the same list
  - Full slabs are not inserted in any list
  - Partial slabs and empty slabs are in the same list
- Try to reduce the memory overhead
- Goal: better scalability on many-core systems
- Some of the SLUB improvements have been ported to SLAB

The Object Cache

- `struct kmem_cache`, from `include/linux/slub_def.h`
  - Similar to the SLAB `kmem_cache`, but simpler
  - Also, the per-CPU free objects cache is implemented as a (lockless!) list (not an array)
  - SLAB uses the Linux "percpu" thing, that disables preemption
- Single slabs list (`partial`): see `kmem_cache_node` in `mm/slab.h`

Example: Object Allocation

- `kmem_cache_alloc()`, defined in `mm/slub.c` invokes `slab_alloc()`, which invokes `slab_alloc_node()`
- `slab_alloc_node()` gets first object from per-CPU cache`->freelist` and updates `freelist`
  - Lockless operation: if the list changed in the meanwhile, redo
- If there are no objects in freelist, invokes `__slab_alloc()`

Refilling the per-CPU Cache

- `__slab_alloc()` is invoked when the per-CPU free objects list (`freelist`) is empty
- `__slab_alloc()` invokes `new_slab_objects()` which invokes `get_partial()`
  - To get a slab from the partial list
- If `get_partial()` fails (no slabs in the partial list), `new_slab()` invokes `allocate_slab()` which invokes `alloc_slab_page()` which invokes `alloc_pages()`