Linux Virtual Memory

Luca Abeni
`luca.abeni@santannapisa.it`

May 20, 2020

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 invokes 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()`

Generic Allocations from slabs

- Slab-based allocators are good for creating caches of “memory objects”
  - All the ojects of a cache have the same size
  - Size declared when creating the cache
- So, how does a generic `kmalloc()` work?
  - Isn’t it based on the slab allocator?
- It uses multiple caches, for objects of different sizes!

kmalloc Caches

- At boot time, multiple `kmalloc-*` caches are created
  - For objects of size 8 bytes, 16 bytes, 32 bytes, 64 bytes, 96 bytes, ...
  - From 256 bytes to 8 kilobytes, only powers of 2
- When `kmalloc()` is used to allocate an amount `s` of memory, find the `kmalloc-` object with size immediately larger than `s`
- See `__kmalloc()` in `mm/slab.c` or `mm/slub.c`
  - For SLAB, `__do_kmalloc()`

kmalloc Details

- If the slab allocator must be used, `kmalloc()` invokes `kmalloc_slab()` to find the correct cache
  - A `kmalloc-` cache containing objects that are large enough
  - See `mm/slab_common.c::kmalloc_slab()`
- For $s \leq 192$, it uses a `size_index` array
- After finding a cache, `slab_alloc()` is invoked
  - See details about SLAB and SLUB

Again on vmalloc

- As mentioned, `vmalloc()` can allocate virtual memory
  - Not contiguous in physical memory
  - Notice: it is memory *for kernel usage*
  - Not in a specific process virtual address space
- Can work for kernel threads too (see later)
- It allocates both a virtual memory fragment and the corresponding physical memory pages
  - Need to modify the default linear mapping
- Memory allocated in a specific range of virtual addresses
  - From `VMALLOC_START` to `VMALLOC_END`
  - vmalloc address space

Basic vmalloc Idea

- In theory, the `vmalloc()` behaviour is not difficult to understand/describe
  - Search for a suitable virtual memory fragment (in the reserved range)
  - Compute how many pages of memory are needed
  - Allocate the physical pages one-by-one, storing them in an array
  - Map the physical pages in virtual memory
- As usual, the devil is in the details...
- Some data structures are needed to store `vmalloc()` information
  - Allocated from slab caches or with `kmalloc`

vmalloc Data Structures

- Defined in `include/linux/vmalloc.h`
  - `struct vmap_area`: describes the memory fragment in virtual memory (`va_start` and `va_end`)
  - `struct vm_struct`: describes how phisical pages are mapped in the virtual memory area
- They are stored in lists and rb trees
- A `vmap_area` contains a pointer to its `vm_struct`
- A `vm_struct` is actually a simplified version of the `mm_struct` describing the virtual address space of a task

Example: Allocation

- Virtual memory allocation is performed by invoking `vmalloc()`
- `vmalloc()` invokes `__vmalloc_node_flags()`, that invokes `__vmalloc_node()` ending up in `__vmalloc_node_range()`
- `__vmalloc_node_range()` rounds up the memory size to a multiple of a page, then invokes `__get_vm_area_node()`, then inovkes __vmalloc_area_node()
  - `__get_vm_area_node()` allocates and initializes `vmap_area` and `vm_struct`
  - __vmalloc_area_node() takes care of actually allocating and mapping the physical pages

Virtual Memory Area Computation

- `__get_vm_area_node()` allocates `vm_struct` (using `kmalloc()`
- Then, allocates and fills `vmap_area` (`alloc_vmap_area()`)
  - `vmap_area` is allocated from a dedicated slab cache
  - Then, it is initialized with the correct `va_start` and `va_end` values
  - And it is inserted in a list of used memory areas
- Then, initializes `vm_struct` with the data from `vmap_area` and sets the `vm` pointer in `vmap_area` (`setup_vmalloc_vm()`)

Physical Pages Allocation

- __vmalloc_area_node() allocates the physical pages for the virtual memory area that has been allocated
- First of all, it allocates an array of `struct page *`
  - Funny recursive allocation (can invoke `__vmalloc_node()`...
  - Fills the `pages` and `nr_pages` fields of `vm_struct`
- Then, allocates all the pages in a for loop
  - Uses `alloc_page()` or `alloc_pages_node()` (with order 0!)
- Finally, maps the allocated physical pages in the virtual memory area (`map_vm_area()`)

Process Address Spaces

- Every user-space process has a private *virtual address space*
  - It contains only a subset of all the possible addresses
  - The other addresses are used for the kernel address space — shared by all processes, but non accessible from user-space
- The kernel address space uses a linear mapping
  - No need to describe it in any data structure
  - Exception: vmalloc address space
- The address space of a process is described by `struct mm_struct` (defined in `include/linux/mm_types.h`)

Virtual Memory Regions

- The virtual address space of a process is composed by multiple *memory regions*
  - A memory region for each segment (code, data, bss, ...)
  - The heap is also a memory region
- Memory regions are page-aligned
- Each memory region is described by a `struct vm_area_struct` (defined in `include/linux/mm_types.h`)
  - Organized in lists and rb trees
  - Contains a link to its address space (`struct mm_struct * vm_mm`)
- The `mmap()` system call can create a new region...

Example: the Heap

- `malloc()` is not a system call: it is a library call
  - Implemented in the standard C library (example: `glibc`)
- The standarc C library allocates memory from the heap
  - Remember? The heap is one of the memory regions of the proces...
- What to do when the heap is empty?
  - The standard C library cannot allocate memory anymore...
  - ...So, it must *grow the heap*
  - Done by invoking a system call: `brk()`

Growing the Heap

- `brk()` system call (`do_brk()`: changes the heap size
  - Technically, it changes the "*program break*" (end of the data segment)
  - Increasing the program break allows to grow the heap by adding more virtual memory pages to this virtual memory region...
- No physical pages are actually allocated!
- Physical pages are allocated only on page faults
  - *Lazy* memory allocation
  - So, do not search for `alloc_page()` in the `do_brk()` call chain...

Page Fault Hanling

- An access to a virtual memory page which is not mapped in physical memory generates a page fault
  - This also happens on write accesses to read-only pages...
  - ...Or in case of violations to page permissions
- Page faults handling is architecture-dependent
  - See, for example, `arch/x86/mm/fault.c::do_page_fault()`
  - It accesses architecture-specific registers to get the faulting address
  - It looks at the current task to get the `mm_struct` structure
- Then, it invokes `handle_mm_fault()`

Architecture Independent Handler

- `mm/memory.c::handle_mm_fault()` receives the virtual memory area containing the faulting address, the address and some flags
- `handle_mm_fault()` ends up invoking `handle_pte_fault()`
  - For a "regular" memory page, ends up invoking `do_anonymous_page()`
- `do_anonymous_page()` ends up in `alloc_pages()` (with order 0)
  - Through `alloc_zeroed_user_highpage_movable()`, remapped in `alloc_page_vma()` $\rightarrow$ `alloc_pages_vma()` with order 0 $\rightarrow$ `alloc_pages()` (for no NUMA)
  - Only when writing to the page for the first time