Linux Memory Management

Luca Abeni
`luca.abeni@santannapisa.it`

May 20, 2020

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