Linux Virtual Memory

Luca Abeni
luca.abeni@santannapisa.it

December 11, 2023

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:exc_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()`, calling `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

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()`)