Advanced Operating Systems (labs)

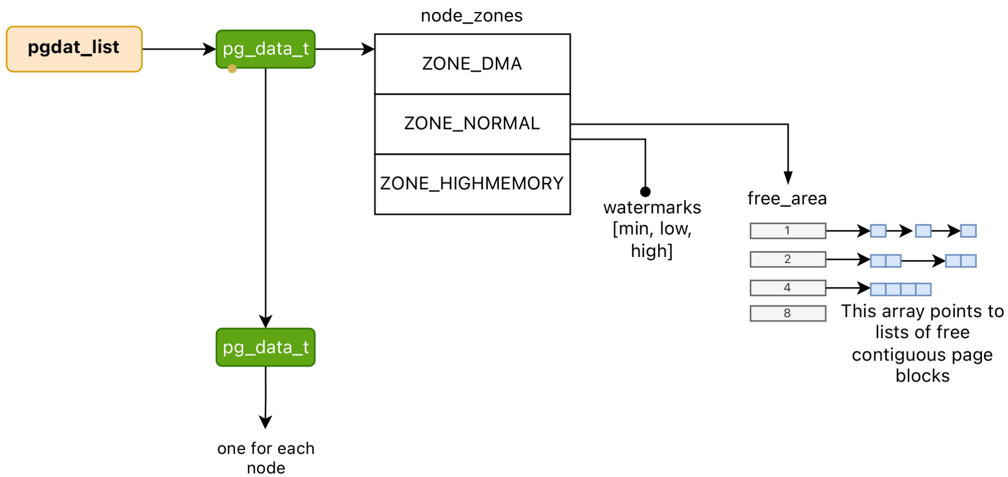
Vittorio Zaccaria | Politecnico di Milano | '24/25

Memory allocation in Linux

Preliminaries - macros

```
#define PN(x) ((void *)((unsigned long long)(x) >> PAGE_SHIFT))
```

Zones



We are going first to print some information around the current Zones in the current node (`print_zones()`).

```
[   26.081186] memalloc: loading out-of-tree module taints kernel.
[   26.195561] Memory Zones for NUMA Node 0:
[   26.197065] Zone 0 - Start PPN: 0x1, End PPN: 0x1000
[   26.200613] Zone 1 - Start PPN: 0x1000, End PPN: 0x7fe0
```

Zones

Recall:

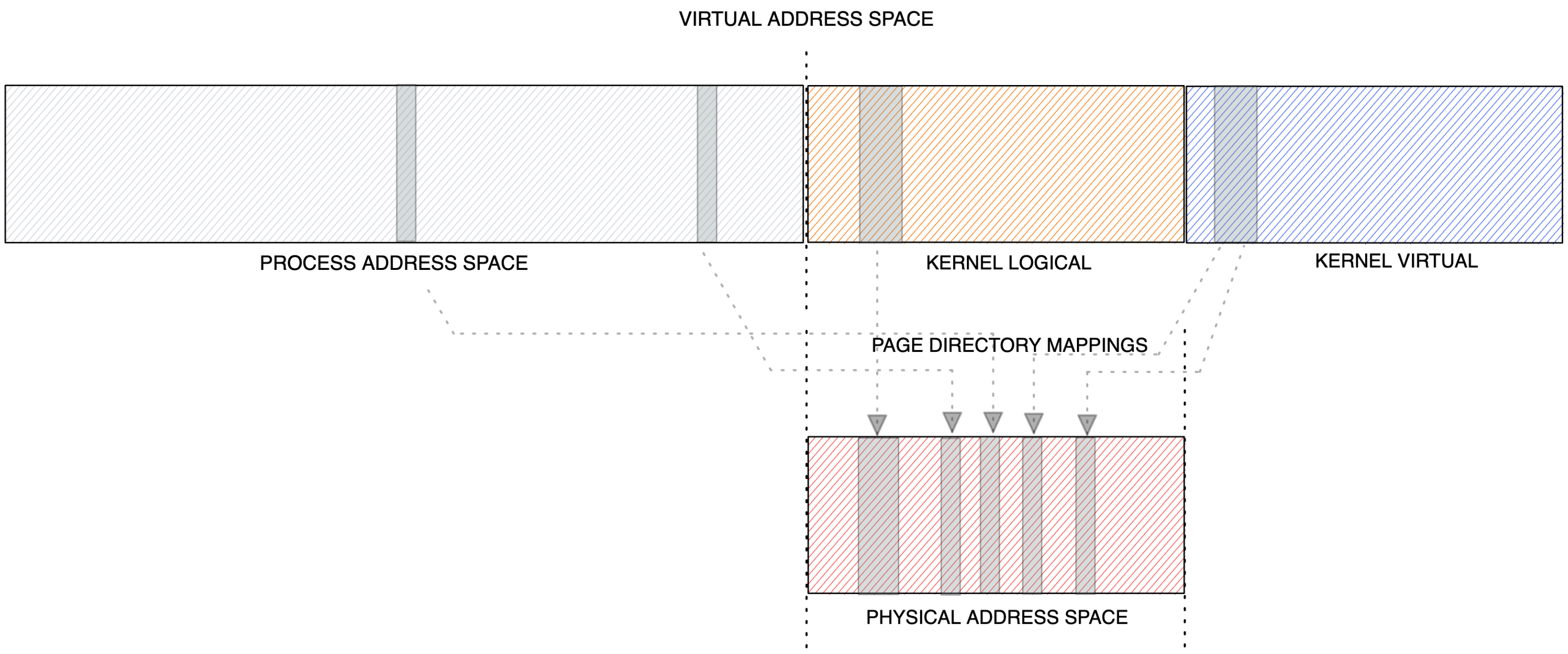
- Zone 0 - Start PPN: 0x1, End PPN: 0x1000. Corresponds to Zone `DMA` which is the low 16 MBytes of memory. At this point it exists for historical reasons; there is legacy x86 hardware that could only do DMA into this area of physical memory.
- Zone 1 - Start PPN: 0x1000, End PPN: 0x7fe0. Corresponds to Zone `DMA32`. `DMA32` exists only in 64-bit Linux; it is the low 4 GBytes of memory, more or less. It exists because the transition to large memory 64-bit machines has created a class of hardware that can only do DMA to the low 4 GBytes of memory.
- `Normal`, On 64-bit machines, it is all RAM from 4GB or so on upwards. Here there is no such a zone.

Cross-checking `/proc/iomem`

If you print the current **physical** memory mappings you can see that Zones cover only a part of the usable addresses. Even Zone 0 has few reserved addresses.

```
/ # cat /proc/iomem
00000000-00000fff : Reserved
00001000-0009fbff : System RAM       ↑
0009fc00-0009ffff : Reserved         |
000a0000-000bffff : PCI Bus 0000:00  |
000c0000-000c99ff : Video ROM.       |
000ca000-000cadff : Adapter ROM      | Zone 0 (DMA)
000cb000-000cb5ff : Adapter ROM      |
000f0000-000fffff : Reserved         |
  000f0000-000fffff : System ROM     |
00100000-07fdffff : System RAM       ↓ ↑ [., 0x00ffffff. ] Zone 0 then
  05400000-0620397f : Kernel code      | [0x01000000, ...] Zone 1
  06400000-0679bfff : Kernel rodata.   |
  06800000-06a88d7f : Kernel data.     | Zone 1 (DMA32)
  0707a000-071fffff : Kernel bss.      ↓
07fe0000-07ffffff : Reserved
08000000-febfffff : PCI Bus 0000:00
  fd000000-fdffffff : 0000:00:02.0
  feb00000-feb7ffff : 0000:00:03.0
  feb80000-feb9ffff : 0000:00:03.0
    feb80000-feb9ffff : e1000
  febb0000-febb0fff : 0000:00:02.0
fec00000-fec003ff : IOAPIC 0
fed00000-fed003ff : HPET 0
  fed00000-fed003ff : PNP0103:00
fee00000-fee00fff : Local APIC
fffc0000-ffffffff : Reserved
100000000-17fffffff : PCI Bus 0000:00
```

Kernel logical and virtual space



> The kernel logical/virtual address space

We are going to show the current kernel logical and virtual AS :

```
[   26.202718] Kernel logical VPN: 000ffff94c080000
[   26.205276] Kernel virtual (VPN - VPN): 000ffffb7c180000 - 000ffffd7c17ffff
```

Kernel memory allocation

Then we are going to use kmalloc and vmalloc to show the corresponding page numbers allocated (`alloc_kmalloc`, `alloc_vmalloc`):

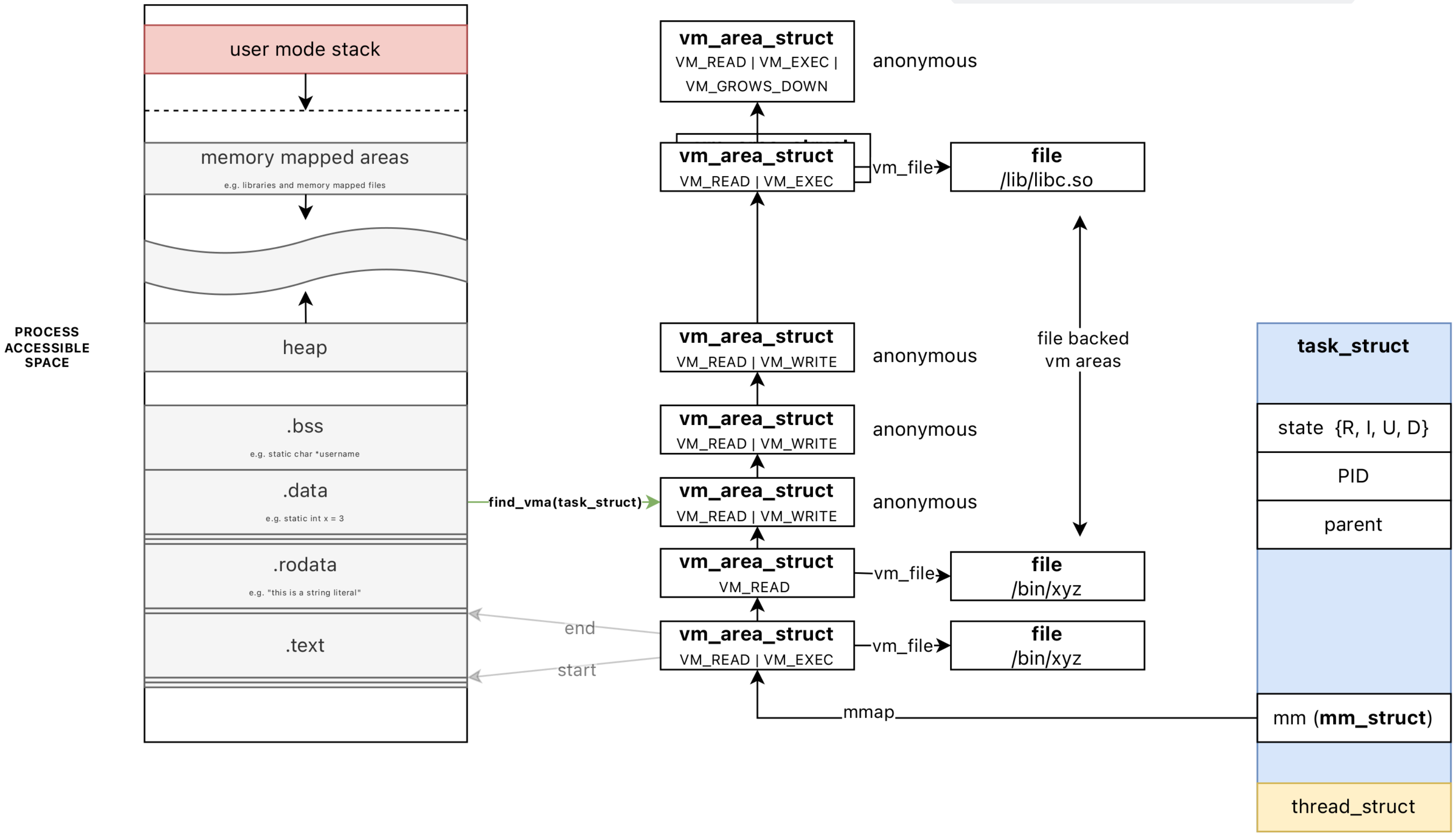
```
[   26.208774] kmalloc - VPN: 000ffff94c081ff8 -> PPN: 0000000000001ff8
[   26.208774]
[   26.215448] vmalloc - VPN: 000ffffb7c180269 -> PPN: 00000000000020fe
[   26.217798] vmalloc - VPN: 000ffffb7c18026a -> PPN: 0000000000002108
[   26.218680] vmalloc - VPN: 000ffffb7c18026b -> PPN: 00000000000021a1
[   26.219391] vmalloc - VPN: 000ffffb7c18026c -> PPN: 00000000000021a2
```

This shows a few things:

- kmalloc - VPN 000ffff94c081ff8 corresponds to PPN 1ff8, a testimony of the fact that all PPN pages are directly mapped from 000ffff94c080000, i.e. the start of the kernel logical addresses
- vmalloc ppns are not necessarily contiguous

Scanning user space VMAs

We are going to scan the current process VMA list (`print_proc_info()`):



Scanning user space VMAs

```
[   28.129124] Current process insmod
[   28.129807] VMA: 0x400000 - 0x401000 R
[   28.130974] VMA: 0x401000 - 0x5f2000 R
[   28.131695] VMA: 0x5f2000 - 0x67e000 R
[   28.132067] VMA: 0x67f000 - 0x686000 R
[   28.132179] VMA: 0x686000 - 0x689000 R
[   28.133118] VMA: 0x689000 - 0x68c000 R
[   28.133443] VMA: 0x174a000 - 0x176d000 R
[   28.133805] VMA: 0x7fca57fe5000 - 0x7fca58033000 R
...
```

The `copy_to/from_user` function

- We are going to show two functions that will become handy to copy to and from userspace from your modules/drivers.
- The `copy_from_user` and `copy_to_user` functions are integral components of the Linux kernel, facilitating secure data transfer between user space and kernel space.
- Both functions are **special** in the sense that, if a crash happens within them (e.g. invalid address) they do not crash the kernel. **They return the n. bytes they weren't able to read/write**.
- Here we are randomly sampling the VMAs of the current process:

```
[   28.140845] We survived...accessing 00000000002ff8ae, read 0 bytes
[   28.142453] We survived...accessing 000000000027e26a, read 0 bytes
[   28.143706] We survived...accessing 0000000000564b8f, read 700 bytes
[   28.144469] We survived...accessing 000000000070a647, read 0 bytes
....
```

The SLUB allocator

- Here we are showing how to create a kernel cache for your own data-structure, by specifying also a constructor `my_struct_constructor` (see function `build_and_fill_kmem_cache(void)`)
- Note that the number of active objects for which `my_struct_constructor` was invoked is higher than the one we allocated with `kmem_cache_alloc`. This is normal as the kernel adopts a speculative heuristics and fills up allocated slabs with active objects.

```
[   28.177397] my_struct_constructor: 1
[   28.177884] my_struct_constructor: 2
[   28.178648] my_struct_constructor: 3
[   28.178980] my_struct_constructor: 4
[   28.179675] my_struct_constructor: 5
...
[   28.242735] my_struct_constructor: 127
[   28.243285] my_struct_constructor: 128


[   28.243939] kmem_cache_alloc: 0
[   28.244748] kmem_cache_alloc: 1
[   28.245427] kmem_cache_alloc: 2
[   28.245813] kmem_cache_alloc: 3
...
[   28.252946] kmem_cache_alloc: 18
```