PB173 – Ovladače jádra – Linux

V. Paměť

Jiri Slaby

ITI, Fakulta informatiky

15. 10. 2013

Práce s pamětí

LDD3 kap. 8 (zastaralá část o bootmem)
Understanding the Linux Virtual Memory Manager (zastaralá)

- Organizace paměti
- Alokace paměti

Část I

Organizace paměti

Organizace paměti

2 oddělené světy
(tentokrát z pohledu HW)

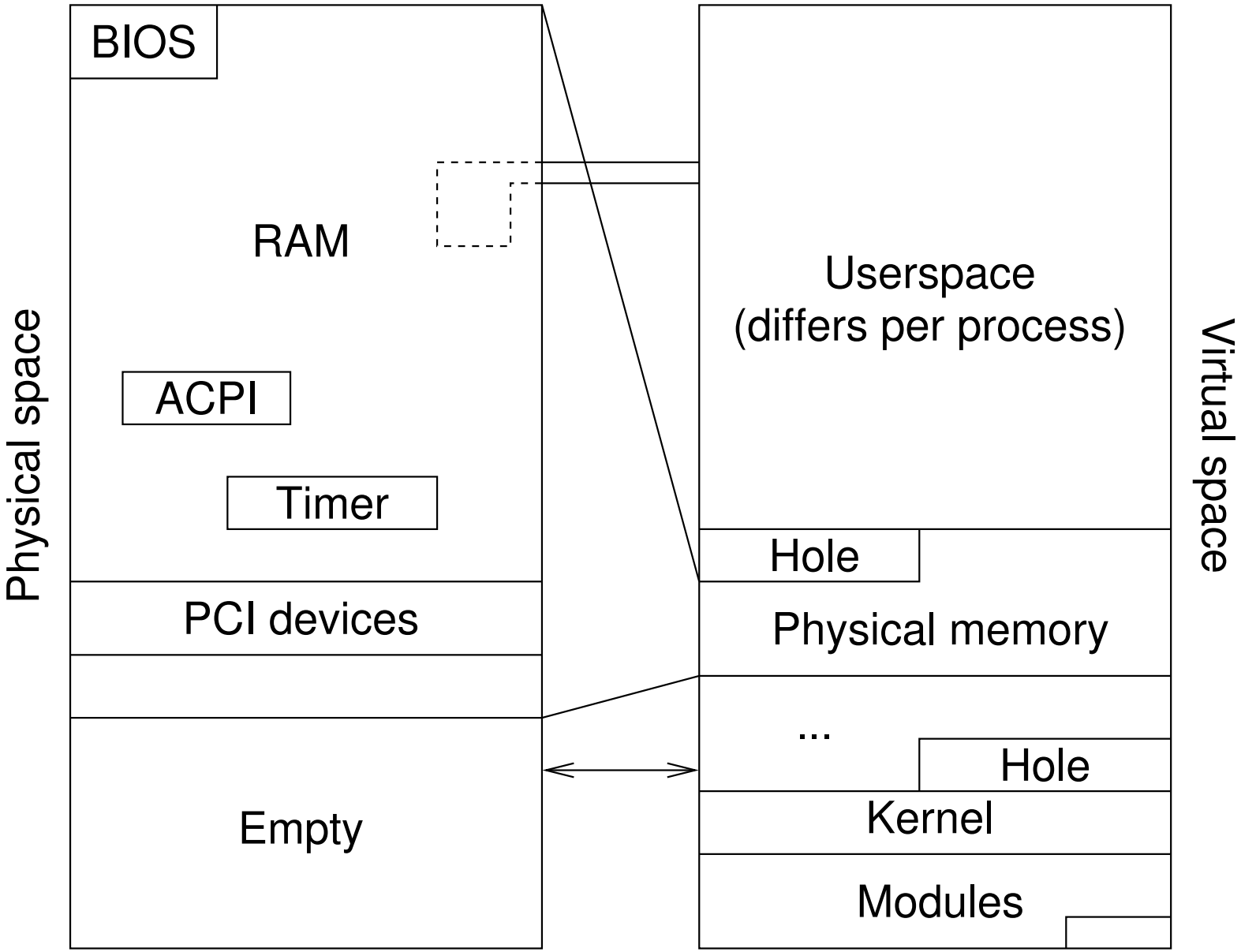
Fyzický prostor

- Rozložení určuje BIOS (popř. ekvivalenty – (U)EFI, Qemu)
- x86: `dmesg|grep BIOS-e820`

Virtuální prostor (paměť)

- Rozložení si určuje OS
  - Různé na každé architektuře
- Dané mapování mezi FP a VP
- Nutná podpora HW (MMU), jinak mapování 1:1

Organizace paměti – graficky



Fyzický prostor

- To, co je za MMU
- RAM, časovače, řadiče přerušení, ACPI tabulky, zařízení
- Co kde je $\Rightarrow$ otázka na BIOS/(U)EFI/...
- Jak který prostor vypadá $\Rightarrow$ specifikace bridge, ACPI, PCI, ...
- I hierarchický prostor
  - Např. rozsah `100–1ff` $\rightarrow$ ChipXY
    - ChipXY směruje dále:
    - Adresa `100–18f` $\rightarrow$ řadič klávesnice
    - Adresa `190–1ff` $\rightarrow$ časovač

RAM

- Reprezentována rozsahy lineárních (fyzických) adres
  - Rozsahy kvůli vloženým prostorům zařízení (e820)
- Cache (některá zařízení také)

Úkol

Práce s fyzickou adresou – přečtení signatury a délky nějaké ACPI tabulky z fyzické adresy

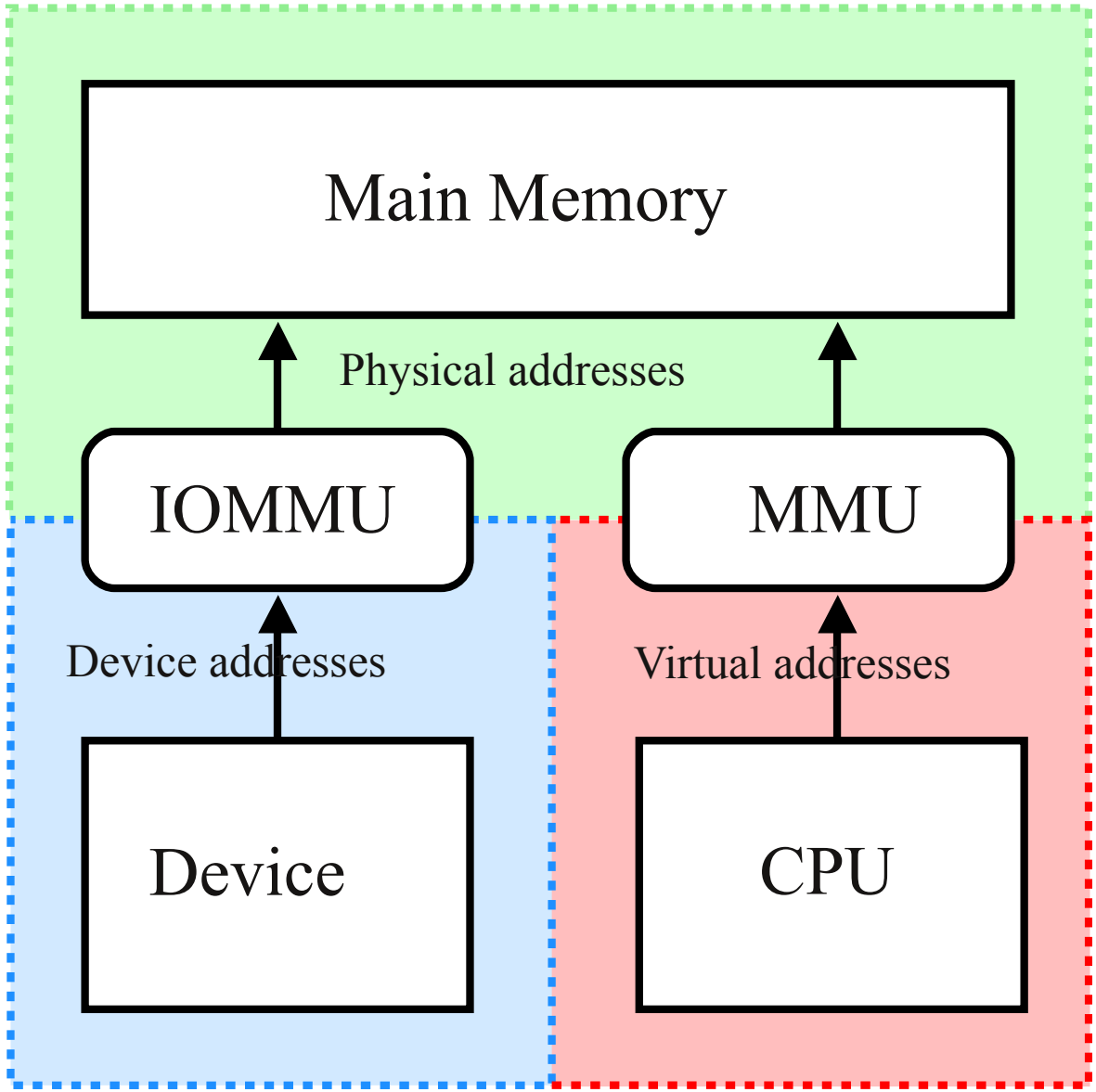
1. Najít fyzickou adresu nějaké ACPI tabulky v `dmesg`
   - Řádky formátu `ACPI: XSDT 00000000af5b0100 00064`
   - Jeden si zvolit (kromě `RSDP`)
   - Zapamatovat adresu (`0x00000000af5b0100`)
2. Získat virtuální adresu (`u32 *virt = ioremap_cache(phys, 8)`)
3. Vypsat signaturu (text, 4B – `virt[0]`)
4. Vypsat délku tabulky (4B – `virt[1]`)
5. Odmapovat (`iounmap(virt)`)
6. Porovnejte délku s výpisem (tj. `0x00064` nahoře)
7. Můžete si přemapovat a vypsat celou tabulku a porovnat položky se specifikací ACPI

Pozn.: na starších jádrech se výpis v dmesg liší

Virtuální prostor

- To, co vidí CPU (a překládá MMU)
  - I 1:1 – start systému, noMMU, . . .
- OS „rozdělí” FP na stránky
  - Index stránky je tzv. *page frame number* (pfn)
- OS namapuje stránky do VP
  - A může přidat „imaginarní” stránky např. na disku (swap)
    - Programy vidí více „paměti”
  - Toto mapování čte MMU (ví odkud)
- Každá stránka má svoji `struct page`
  - Informace o stránce (nikdy nevyswapovat, špinavá, počet uživatelů)

Grafické shrnutí



Zdroj: wikipedia

Operace s adresami

- `linux/mm.h`, `linux/io.h`
- Např. `void *virt = __get_free_page`

Fyzická adresa

- `phys_addr_t phys = virt_to_phys(virt)`
- `void *virt = phys_to_virt(phys)`

Struktura `page`

- `struct page *page = virt_to_page(virt)`
- `void *virt = page_to_virt(page)`

PFN

- `unsigned long pfn = page_to_pfn(virt_to_page(virt))`
- `void *virt = pfn_to_virt(pfn)`
- `unsigned long pfn = phys >> PAGE_SHIFT`
- `phys_addr_t phys = pfn << PAGE_SHIFT`

Úkol

Práce s virtuální adresou

1. `linux/mm.h`
2. Naalokujte stránku (`virt = __get_free_page(GFP_KERNEL)`)
3. Nakopírujte do ní řetězec
4. Zjistěte fyzickou adresu (`phys = virt_to_phys(virt)`)
5. Zjistěte adresu struct page (`page = virt_to_page(virt)`)
6. Zarezervujte stránku (`SetPageReserved(page)`)
7. Přemapujte fyzickou stránku (`map = ioremap(phys, ...)`)
   - Varování v `dmesg` ignorujte
8. Vypište `virt`, `phys`, `page`, `map` a `page_to_pfn(page)`
9. Vypište obsah (`%s`) `virt` a `map`
10. Změňte obsah `map` a vypište obsah `virt`
11. `iounmap`, `ClearPageReserved` a `free_page`
12. Zkonzultujte s `Documentation/x86/x86_64/mm.txt`

Část II

Alokace paměti

Alokace paměti

Základní typy alokací

- Malé bloky až souvisle po stránkách (`kmalloc`)
- Souvisle po stránkách (`__get_free_pages`)
- Nesouvisle po stránkách a namapovat (`vmalloc`)

GFP_*

- Parametr pro alokátory
- `GFP_ATOMIC` – nespi (např. uvnitř spinlocků)
- `GFP_TEMPORARY` – uvolním během pár příštích instrukcí
- `GFP_KERNEL` – obyčejné alokace všude jinde
- a další: `GFP_NOFS`, `GFP_NOIO`, ...

SLAB alokátor

- Vnitřně alokuje po stránkách
  - Stránka má velikost `PAGE_SIZE` (x86: 4K)
- Rozděluje stránky na menší kusy
  - Výhýbá se fragmentaci paměti
- `linux/slab.h`
- `kmalloc`, `kzalloc` (= `kmalloc` + `memset`), `kfree`

Úkol

Alokace pomocí `kmalloc`

1. Alokujte 32 stránek (jednou alokací)
2. Vypište jejich pfn
   - Skákejte po `PAGE_SIZE`
   - Použijte `virt_to_page` a `page_to_pfn`

__get_free_pages

Stránkový alokátor

- Musí najít souvislý blok volných stránek
  - Může být problém najít větší bloky (řádu $>= 2$) kvůli fragmentaci
- Rychlejší než `kmalloc`
- Používá tzv. *buddy* systém
  - Snižuje fragmentaci
  - Podrobnosti viz wiki
- `linux/mm.h`
- Velikost alokace se udává řádem, `order` (2^{order} = počet stránek)
  - `get_order`, `MAX_ORDER` (na x86 = 11 = 8M)
- `__get_free_pages`, `free_pages`
- `__get_free_page`, `free_page` (order 0)

Úkol: alokace 32 stránek a výpis pfn

Virtuální alokátor

- Použitím podobný `kmalloc`
- Neatomický (vnitřně používá `GFP_KERNEL`)
- Maximální velikost alokace daná omezením architektury
  - x86: 128 MiB (lze zvýšit parametrem, ne o moc)
  - x86_64: 30 TiB
- Alokuje po stránkách a mapuje je souvisle
  - Fyzické stránky jsou umístěné různě po RAM
  - Dražší
- `linux/vmalloc.h`
- `vmalloc`, `vfree`
- Odlišná práce s adresami (*POZOR*)
  - `vmalloc_to_page`, `vmalloc_to_pfn`

Úkol: alokace 32 stránek a výpis pfn

Poznámky na závěr

- Vše, co může číst uživatel se musí nejprve vymazat
  - `kzalloc`, `get_zeroed_page`, `vzalloc`
  - Nejlépe vymazat všechno, co alokuji a není kritické
- Používat správné `GFP_*`
  - `GFP_ATOMIC` funguje všude, ale ubírá vzácné prostředky

Úkol

Alokace paměti

1. Zjistěte, kolikrát za sebou lze naalokovat pamět:
   - řádu 10 jako `GFP_ATOMIC`
   - řádu 10 jako `GFP_ATOMIC` (znovu po 5 vteřinách)
   - řádu 10 jako `GFP_KERNEL`
   - řádu 10 jako `GFP_KERNEL` po uvolnění předešlé paměti
   - Pozn.: $2^{10} \cdot 4096 = 4\,\mathrm{M}$, jak velké pole na ukazatele?
2. Zkuste ve smyčce alokovat paměť s řádem $\leq$ `PAGE_ALLOC_COSTLY_ORDER`, co se stane? Přijde OOM killer...

Demo: pb173/05