Metody připojování periferií

BI-MPP

Přednáška 9

Ing. Miroslav Skrbek, Ph.D.

Katedra číslicového návrhu
Fakulta informačních technologií
České vysoké učení technické v Praze

ZS2021/22

Evropský sociální fond
Praha & EU: Investujeme do vaší budoucnosti

Agenda

- Ovladače USB zařízení v Linuxu
- Windows Driver Model

Literatura

- Gook, M.: Hardwarová rozhraní – Průvodce programátora. Computer Press, Brno 2006. ISBN 80-251-1019-2
- Corbet, J., Rubini A., Kroah-Hartman, G.: Linux Device Drivers. Third Edition. O'Reilly Media 2005. ISBN: 0-596-00590-3.
- WDK Documentation, Windows Drivers Kits, Microsoft Corporation, 2009.

URB

- USB request block
- Datová struktura reprezentující USB přenosy
- Používá se pro posílání nebo přijímání dat z daného endpointu
- Zajišťuje asynchronní přenosy
- Obsahuje
  - Číslo endpointu
  - Délku přenosu
  - Adresu bufferu (pro příjem nebo odeslání dat)
  - Stav přenosu
  - Adresu dokončovací rutiny, která se volá po ukončení přenosu
- Vytváří se a ruší
  - struct urb *usb_alloc_urb(int iso_packets, int mem_flags);
  - void usb_free_urb(struct urb *urb);

Příprava URB

- Pro vyplnění URB slouží funkce
  - Interrupt transfers

```
void usb_fill_int_urb(struct urb *urb, struct
usb_device *dev,unsigned int pipe, void*
transfer_buffer, int buffer_length, usb_complete_t
complete, void *context, int interval);
```

  - Bulk transfers

```
void usb_fill_bulk_urb(struct urb *urb, struct
usb_device *dev, unsigned int pipe, void*
transfer_buffer, int buffer_length, usb_complete_t
complete, void *context);
```

  - Control transfers

```
void usb_fill_control_urb(struct urb *urb, struct
usb_device *dev, unsigned int pipe, unsigned char
*setup_packet, void *transfer_buffer, int
buffer_length
```

Odeslání URB ke zpracování

- **`int usb_submit_urb(struct urb *urb, int mem_flags);`**
  - Parametr urb je ukazatel na URB
  - Parametr mem_flags se nastavuje podle kontextu, ve kterém funkce je použita
    - GP_ATOMIC
    - GP_NOIO
    - GP_KERNEL
- Zpracování URB lze zastavit funkcemi
  - int usb_kill_urb(struct urb *urb);
  - int usb_unlink_urb(struct urb *urb); (nečeká na dokončení)

Informace o podporovaných zařízeních driverem

```
static struct usb_device_id mydev_table [ ] = {
{ USB_DEVICE(USB_MYDEV_VENDOR_ID, USB_MYDEV_PRODUCT_ID) },
{ } /* ukončující položka */
};
MODULE_DEVICE_TABLE (usb, mydev_table);
```

Pokud driver podporuje více zařízení s různými VID a PID, pak má tabulka více položek. Tabulka se používá pro vyhledání driveru pro dané zařízení.

Registrace USB driveru

```
static struct usb_driver my_driver = {
.owner = THIS_MODULE,
.name = „myusbdev",
.id_table = mydev_table,
.probe = mydev_probe,
.disconnect = mydev_disconnect,
};
```

Registrace driveru uvnitř inicializační funkce modulu

```
result = usb_register(&mydev_driver);
if (result)
   err("usb_register failed. Error number %d",
result);
```

Probe

Funkce probe slouží k ověření zařízení z hlediska schopnosti driveru ovládat toto zařízení

Vytváří vnitřní lokální struktury pro konkrétní (instanci) zařízení a registruje ho v jádře

Vrací 1 pokud je zařízení úspěšně registrováno

Registrace USB zařízení v Probe

```
retval = usb_register_dev(interface, &mydev_class);
if (retval) {
        err("Not able to get a minor for this device.");
        usb_set_intfdata(interface, NULL);
        goto error;
}
```

Přenosy bez URB

```
int usb_bulk_msg(struct usb_device *usb_dev, unsigned int
 pipe,void *data, int len, int *actual_length,
 int  timeout);

int usb_control_msg(struct usb_device *dev,
  unsigned int pipe,__u8 request, __u8 requesttype,
  __u16 value, __u16 index, void *data, __u16 size,
  int timeout);
```

Zjednodušují psaní driveru, pro realizaci přenosu stačí jedno volání funkce

usb_bulk_msg je pro blokové přenosy a usb_control_msg je pro řídící přenosy

Windows Driver Model

- Model ovladačů pro OS Windows, který podporuje Plug&Play zařízení
- K dispozici je od Windows 98
- Vývoj ovladačů probíhá v jazyce C (včetně knihovní podpory)
- Pro vývoj ovladačů je k dispozici Windows Driver Kit (WDK)

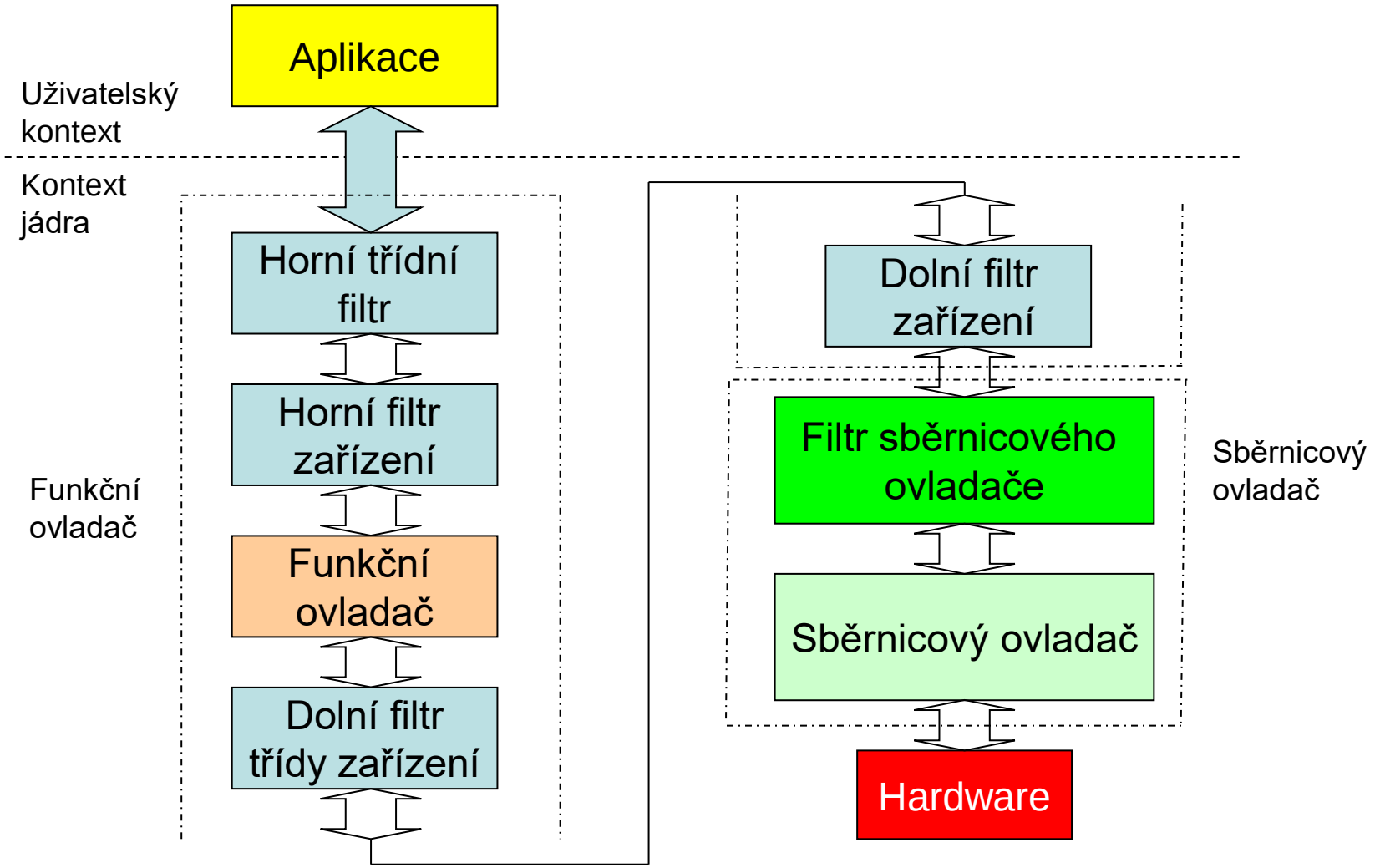
Rozdělení ovladačů dle kontextu

- Kernel Drivers
  - Jsou ovladači v kontextu jádra
  - Mohou přistupovat přímo na hardware a využívat přímého přístupu do paměti
  - Složitější vývoj a ne tak komfortní ladění
- User Mode Drivers
  - Jsou ovlači v uživatelském kontextu
  - Mají formu DLL knihovny
  - Nacházejí v horních patrech hierarchie ovladačů pro dané zařízení
  - Tento typ driverů využívají např. USB zařízení

Rozdělení ovladačů dle funkce

- Sběrnicové ovladače (Bus Drivers)
  - Obsluhují řadiče sběrnic, sběrnicové adaptéry nebo můstky
  - Příkladem jsou sběrnicové ovladače pro PCI, SCSI a USB
  - Najdeme je na nejnižší úrovni v hierarchii
  - Typicky zajišťují enumeraci (identifikaci a evidenci) zařízení připojených na sběrnici a konfiguraci těchto zařízení
- Funkční ovladače (Function Drivers)
  - Ovladač konkrétního typu zařízení (myš, tablet, …)
  - Obvykle poskytován výrobcem zařízení
- Filtry (Filter Drivers)
  - "průchozí" ovladače vkládané mezi ovladače
  - Používají se pro logovací nebo bezpečnostní funkce

Hierarchie ovladačů



Vývoj ovladačů

- Windows Driver Model
  - Původní, k dispozici od Windows 98
  - Knihovní podpora pouze základních funkcionalit
  - Typické funkcionality ovladače je nutno implementovat v každém ovladači znovu a znovu
  - Náročné na vývoj a znalosti
- Kernel-Mode Driver Framework (KMDF)
  - Knihovna, která poskytuje sadu funkcí na vyšší úrovni, které zjednodušují vývoj ovladačů
  - Typické funkcionality jsou implementovány v knihovně
  - Knihovna využívá objektový přístup
  - Implementace v jazyce C
  - Snažší vývoj, zvláště pro začátečníky

Objektový přístup (princip)

Jedná se o objektový přístup ve smyslu objektového programování.
Implementován v jazyce C.

Instanční proměnné jsou ve strukturách

```
typedef struct {
    int pocet;
    void (*clean)(PMY_CLASS obj);
} MY_CLASS, *PMY_CLASS;
```

Metody

```
void set(PMY_CLASS obj, int value) {
          obj->pocet = value;
}
```

Destruktor dealokuje instanční proměnné typu reference na objekt

Nová instance třídy

```
PMY_CLASS my_class_create(PMY_CLASS obj, int value, void (*clean)(PMY_CLASS obj)()) {
          PMY_CLASS obj = (PMY_CLASS)calloc(1, sizeof(MY_CLASS));
          return obj;
}
```

Zrušení instance třídy

```
void my_class_destroy(PMY_CLASS *obj) {
          (*obj)->clean(*obj);        // volání callbacku clean pro výmaz referencí
          free(*obj);
          *obj = NULL;
}
```