# PB173 – Ovladače jádra – Linux

## II.

Jiří Slabý

ITI, Fakulta Informatiky

5. 10. 2010

# Důležité informace

- Kolokvium za DÚ
  - DÚ do příštího cvičení
- Login/heslo
  1. Exportované domovské adresáře ⇒ login/login (z FI)
  2. Jinak ⇒ vyvoj/vyvoj
- GIT: `git://decibel.fi.muni.cz/~xslaby/pb173`
  - `git pull --rebase`
- Qemu obraz: `/home/local/centos.img`
  - 2 účty: root/toor, user/user
  - `./qemu_start` z GITu

# Dnešní téma

**Komunikace**
**jádro $\leftrightarrow$ uživatelský prostor**

# Komunikace s uživatelem I.

1. Voláním funkce: system call (syscall)
   - V jádře: tabulka číslo-funkce (`unistd.h`)
   - Skok do jádra speciální instrukcí (x86_64: `syscall`)
     - O skok se stará libc (`fwrite`→`write`→`syscall`→instrukce)
     - Drahá operace (přepnutí kontextu)
   - `syscall(__NR_write, STDOUT_FILENO, "Test\n", 5);`
   - Demo: lxr → `__NR_write`
2. Speciální syscall
   - Bez skoku do jádra, v knihovně linkované jádrem (`vdso.so`)
   - Několik málo jednoduchých funkcí (`gettimeofday`, ...)

**Každá nová funkcionalita = nový syscall**

# Úkol

V userspace: pomocí volání `syscall` spustit `/bin/date`. Předtím vypsat nějakou informaci.

1. `syscall(__NR_write, ..., "Running date\n", ...)`
2. `syscall(__NR_execve, "/bin/date", ...)`

Je třeba znát prototypy funkcí.

- Většinou jsou dokumentované: `man write`, `man execve`
- Jinak použít lxr
  - `sys_fork`
  - `SYSCALL_DEFINE1(rmdir, ...)` → `sys_rmdir`
  - `SYSCALL_DEFINE3(read, ...)` → `sys_read`

# Komunikace s uživatelem II.

3. Speciální soubory v `/dev`
   - Komunikace přes soubor (není nutný nový syscall)
   - Seznam v **`Documentation/devices.txt`** a `/proc/devices`
   - Identifikované jako major a minor čísla
     - Většinou major=ovladač, minor=zařízení (tty: 4, 0-63)
   - Blokové (disky apod.)
     - Komunikace po blocích
     - Nebudeme se jimi zabývat (popsány v LDD)
   - Znakové (ostatní)
     - Komunikace po znacích (bajtech)
     - Viz následující slidy
4. Sockety, roury, . . .

# Znaková zařízení I.

- LDD3 3. a 6. kapitola
- 2-3 kroky
  1. Registrace rozsahu major+minor (`module_init`)
     - `alloc_chrdev_region`, `register_chrdev_region`, `unregister_chrdev_region` (**`linux/fs.h`**)
     - Přidání záznamu do `/proc/devices`
  2. Registrace jednotlivých minorů (PCI, USB, ... `probe`)
     - `cdev_add`, `cdev_del` (**`linux/cdev.h`**)
     - Po odpovídajícím `mknod` lze zařízení používat
  3. Podat zprávu `udev` (vytvoření `/dev/*`) – nepovinné (`probe`)
     - `device_create`, `device_destroy` (**`linux/device.h`**)
     - Předem je potřeba vytvořit `class` (`module_init`)
- Implementuje `open`, `close`, `read`, `write`, `ioctl`, `mmap`, ...

# Znaková zařízení II.

Stačí-li 1 zařízení (1 minor), lze použít vrstvu `misc`

- Dělá všechnu práci z předchozího slidu
- Potřebujeme
  - Seznam implementovaných funkcí (opět)
  - Definici misc zařízení (`struct miscdevice`)
- `misc_register`, `misc_deregister`
- Objeví se v `/proc/misc` a `/dev`
- **`linux/miscdevice.h`**

# Popis funkcí

- `struct file_operations` (**`linux/fs.h`**)
- Seznam funkcí, které chceme obsluhovat
- Parametr pro `cdev_add`, jsou v `miscdevice` apod.

```
struct module *owner; /* = THIS_MODULE */
int (*open)(struct inode *inode, struct file *filp)
ssize_t (*read)(struct file *filp, char __user *buf, size_t count,
    loff_t *offp)
int (*release)(struct inode *inode, struct file *filp);
```

- `filp->private_data` slouží programátorovi (libovolně)
- `offp` slouží programátorovi (k poznamenání průběhu)
- `__user` značí ukazatel od uživatele
- Návratové hodnoty
  - `int` – záporné = `-Echyba`, jinak 0
  - `ssize_t` – záporné = `-Echyba`, jinak počet zpracovaných znaků

# Úkol

Vytvořit misc (znakové) zařízení s obsluhou `open`, `read`, `write`, `release` (tj. close).

1. Definice `struct file_operations`
2. Vytvoření funkcí dle prototypu z `file_operations` (nalézt v lxr)
   - Open a release s nějakým `printk`
   - Read a write prozatím prázdná těla
   - Open a release vracejí 0 (žádná chyba), read též (EOF), write `count` (zapsáno vše)
3. Definice `struct miscdevice`
   - `minor = MISC_DYNAMIC_MINOR`, `name`, `fops`
4. `misc_register/deregister` do `module_init/exit`
5. `make`, `insmod`
6. Vyzkoušení `cat /dev/name` (`name` je z `miscdevice`)

# Uživatelská paměť

- Něco, čemu nelze věřit (NULL, ukazatel to tabulek oprávnění, . . . )
- Nutnost kontroly
- `copy_from_user`, `copy_to_user`
  - „`memcpy`” s kontrolou
  - Vracejí počet NEzkopírovaných znaků (0=OK)
- `get_user`, `put_user`
  - Jen primitiva (`char`, `short`, `int`, `long`)
  - „`var = *(type *)buf`” a „`*(type *)buf = var`” s kontrolou
  - Vracejí 0 nebo chybu (záporná hodnota)
- Definované v **`linux/uaccess.h`**

Demo: pb173/02

# Úkol

Dopsat těla funkcí `read` a `write` tak, aby zpracovávala data.

1. `write` vypíše uživatelský buffer pomocí `printk`
   - `copy_from_user` (chyba = return -EFAULT)
   - Ukončit zkopírovaný řetězec pomocí \0
2. `read` bude vracet ”Ahoj”
   - `copy_to_user`

# Datové typy

- Jádro a proces může běžet s různými bitovými šířkami (32, 64-bit)
- Problém s ukazateli a long proměnnými
  - Jiná délka dat
  - Jiné zarovnání struktur
- Jádro definuje pevné typy (z hlediska počtu bitů)
  - **linux/types.h**
  - `__u8`, `__u16`, `__u32`, `__u64`
  - `__s8`, `__s16`, `__s32`, `__s64`
  - Ukazatele musí být v union s `__u64`

```
struct my {
  unsigned long flags;
  short index;
  void *data;
} my;
```

⇒

```
struct my {
  __u64 flags;
  __s16 index;
  union {
    void *data;
    __u64 filler;
  };
} my;
```

```
read(fd, &my, sizeof(my));
ioctl(fd, DO_SOMETHING, &my);
```