PB173 – Ovladače jádra – Linux
II.

Jiří Slabý

ITI, Fakulta Informatiky

4. 10. 2011

Důležité informace

- Kolokvium za DÚ
  - DÚ do příštího cvičení
- Login/heslo
  - vyvoj/vyvoj
- GIT: `git://github.com/jirislaby/pb173.git`
  - `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 I.

1. Voláním funkce: system call (syscall)
   - Skok do jádra speciální instrukcí (x86_64: `syscall`)
     - O skok se stará libc (`fwrite`→`write`→`syscall`→instrukce)
     - Drahá operace (přepnutí kontextu)
   - 1. parametr `syscall`u je číslo operace
     - V jádře: tabulka číslo-funkce
   - `syscall(__NR_fork)`
   - Demo: lxr → `__NR_fork`
2. Speciální syscall
   - `vdso.so`

Každá nová funkcionalita = nový syscall

Úkol

V userspace: pomocí `syscall` vypsat nějakou informaci

1. `syscall` a `__NR_write`

Je třeba znát prototypy funkcí.

- Většinou jsou dokumentované: `man write`
- Jinak použít lxr

Komunikace 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; viz dále)
  - Definici misc zařízení (`struct miscdevice`)
- `misc_register`, `misc_deregister`
- Objeví se v `/proc/misc` a `/dev`
- **`linux/miscdevice.h`**

Popis funkcí

Seznam funkcí, které chceme obsluhovat

- `struct file_operations` (**`linux/fs.h`**)
- Parametr pro `cdev_add`, součást `miscdevice` apod.

```
ssize_t (*write)(struct file *filp, const char __user *buf, size_t
    count, loff_t *offp)
```

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

Úkol

1. Najděte všechny možné funkce, které lze obsluhovat
   - Najděte `struct file_operations` v lxr
   - Projděte strukturu
2. Najděte možné chybové návratové hodnoty
   - Najděte `EPERM` v lxr
   - Projděte ostatní

Příklad operací

```
struct file_operations my_fops = {
  .owner = THIS_MODULE,
  .open = my_open,
  .write = my_write,
  .release = my_release,
};

int my_open(struct inode *inode, struct file *filp) {
  filp->private_data = __something;
  return 0;
}

ssize_t my_write(struct file *filp, const char __user *buf, size_t
    count, loff_t *offp) {
  /* filp->private_data is __something here */
  if (count == 0)
    return -EINVAL;
  return 0;
}
```

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

Uživatel (32-bit)		Jádro (64-bit)
`data` ... 4 bajty	`struct my { void *data; };`	`data` ... 8 bajtů

Pevné typy v jádře

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);
```