Linux Programozás

Kernel fejlesztés

Konkurencia kezelés

- Atomi műveletek
  - Egyszerű műveletek, nincs szükség szinkronizációra
- Spinlock
  - busy cycle jellegű
  - Rövid szakaszok esetén
  - A szakasz nem tartalmazhat sleep-et
- Semaphore
  - Összetettebb mechanizmus, így erőforrás igénye nagyobb
  - A spinlock megkötései nem érvényesek rá
- Mutex
  - Mint a semaphore, de egyszerűbb

I/O portok kezelése

- Az eszközök jelentős részével az eszközvezérlő I/O portokon keresztül kommunikál a processzor in és out parancsaival.
- Mielőtt az IO portokat használnánk a driverben illik lefoglalni a használt port tartományt.
- A modul eltávolításakor pedig fel kell szabadítanunk a port tartomány foglalást.

I/O port tartomány lefoglalása és felszabadítása

- Lefoglalás:

```
struct resource*
request_region(resource_size_t start,
resource_size_t n, const char *name);
```

- A lefoglalhatóság ellenőrzése:

```
int check_region(resource_size_t start,
resource_size_t n);
```

- Felszabadítás:

```
void release_region(resource_size_t start,
resource_size_t n);
```

- A lefoglalt I/O portok listája:

```
/proc/ioports
```

I/O műveletek

- 8 bites

```
unsigned char inb(int port)
void outb(unsigned char value, int port)
```

- 16 bites

```
unsigned short inw(int port)
void outw(unsigned short value, int port)
```

- 32 bites

```
unsigned long inl(int port)
void outl(unsigned long value, int port)
```

- Mindegyik függvénynek vagy „pause”-os változata: A függvény nevének a végére egy „_p”-t kell írni.

I/O port kezelés példa

Példa: „kbdriver.c”

- Tipp:
  - Ha a „cat” paranccsal megnézzük az eszközállományt, akkor mindig a parancs végi enter lenyomás scan kódját kapjuk.
  - Ellenben ha a „watch -n cat file” paranccsal nézzük akkor monitorozni tudjuk az állapot változását.

Megszakítás kezelés

- Inicializálás:
  - Megszakítás kezelő függvény létrehozása
  - A függvény regisztrációja
- Felszabadítás
  - A regisztráció megszüntetése

A megszakítás kezelő regisztrációja

```
int request_irq(unsigned int irq,
irq_handler_t handler, unsigned long
flags, const char *devname, void
*devid);
```

- A megszakítás kezelő függvény:

```
typedef irqreturn_t (*irq_handler_t)(int
irq, void *devid);
```

A flags mező értékei

- `IRQF_DISABLED`: A gyors interrupt jelzése.
- `IRQF_SHARED`: Annak jelzése, hogy az interruptot megosztjuk más megszakítás kezelőkkel. Ilyenkor általában több hardver használja ugyanazt a megszakítás vonalat.
- `IRQF_SAMPLE_RANDOM`: A megszakítás felhasználható a véletlenszám generáláshoz.
- `IRQF_TIMER`: A megszakítás timer interrupt.

A megszakítás kezelő függvény visszatérési értékei

- `IRQ_HANDLED`: A megszakítást lekezelte a függvény.
- `IRQ_NONE`: A megszakítást nem kezelte a függvény.

A megszakítás kezelésének megszüntetése

```
void free_irq(unsigned int irq, void
*devid);
```

Gyors és lassú interruptok

- Gyors
  - A megszakítások maszkolódnak az adott processzoron.
  - SMP rendszerben a többi processzorok kezelhetnek megszakításokat.
  - Nem javasolt a használata, csak speciális esetekben.
- Lassú
  - A feldolgozás során a többi megszakítás nem maszkolódik.

Megszakítások megosztása

- A megszakításkezelő regisztrálásánál használjuk az `IRQF_SHARED` flaget.
- A `devid` értékének egyedinek kell lennie.
- Ha a lekezelő függvény úgy érzékeli, hogy nem az adott eszköz generálta a megszakítást, akkor `IRQ_NONE` értékkel kell visszatérnie.
- Vigyázzunk a megszakítás letiltásával / engedélyezésével.

Megkötések megszakítás kezelő függvényekre

- Nem használhatunk sleep-es függvényt.
  - Így a `kmalloc`-os allokációt is csak a `GFP_ATOMIC` flaggel végezhetünk.
  - A kernel és a user space között nem mozgathatunk adatokat.
- Törekedjünk a gyorsaságra. A nagy számításokat lehetőleg máshol végezzük.
- Mivel nem processz hívja meg a függvény, ezért a processz specifikus adatok nem elérhetőek.

BH mechanizmus

- Top half
- Bottom half
  - Tasklet
  - Workqueue

Megszakítás példa

Példa: „`kbirq.c`”

Linux eszközvezérlő modell

- A klasszikus eszközvezérlők csak eszközállomány interfészt regisztráltak.
- A modern eszközvezérlők (2.6+) az eszközvezérlő modellt implementálják. Lehet eszközállomány interfészük, de nem szükséges.
- A modell elemei:
  - Busz – PCI, USB, i2c, spi, SCSI, …
    - Eszközök kapcsolódnak hozzá.
  - Eszköz
    - A busz által meghatározott módon kommunikálhatunk vele.
  - Eszközvezérlő
    - Kezeli az egyes eszköz típusokat
  - Osztály – audio, mutató eszközök, hwmon, ...
    - Eszközök csoportosítása funkció szerint

Eszközvezérlő regisztrálása

- Busz specifikus leíró struktúra és regisztrációs függvény.
- Callback függvényeket regisztrálunk az egyes eseményekhez.

```
static struct i2c_driver mitronavr_driver = {
    .driver = {
        .name  = DRVNAME,
    },
    .id_table = mitronavr_ids,
    .probe = mitronavr_probe,
    .remove    = mitronavr_remove,
    .class     = I2C_CLASS_HWMON,
    .detect    = mitronavr_detect,
    .address_list  = normal_i2c,
};
```

Tipikus beállítások

- Azonosítók, ami alapján a buszvezérlő beazonosíthatja az eszközvezérlőt.
- probe:
  - A buszvezérlő az azonosító alapján hívja és részlegesen az eszközvezérlőhöz köti az eszközt.
  - Ellenőrizzük, hogy ténylegesen kezelhető-e az eszköz.
  - Allokáljuk és az eszközhöz rendeljük a leíró adatstruktúrát.
  - Opcionálisan elvégezzük az inicializációkat.
  - Ha sikert jelzünk vissza, akkor a buszvezérlő teljesen hozzáköti az eszközt az eszközvezérlőhöz.
- remove:
  - Az eszköz kötés megszűnésekor hívódik.
  - Elvégezzük a felszabadításokat.

Kommunikáció

- A tényleges kommunikáció az eszközökkel erősen busz specifikus.
- A buszvezérlő nyújt egy API-t, aminek segítségével kommunikálhatunk.

Eszköz attribútumok

- Ha az eszközvezérlőnk követi a Linux ezközvezérlő modelt, akkor a sysfs könyvtárrendszerében megjelenik hozzá egy könyvtár.
- A buszok (USB, PCI, platform, …) regisztráló függvényei gondoskodnak erről.
- A csatlakoztatott eszközökhöz attribútumokat rendelhetünk
  - get / set függvényeket adunk meg
  - sysfs állományt hozunk létre

Példa: „devattr.c”

Az rpi2exp kiegészítő kártya

Eszköz	Típus	Cím	DS
4 LED (GPIO)		22,23,24,25	
4 Nyomógomb (GPIO)		5,6,12,13	
Potméter (I2C)	MCP40D18T	0x2e	Link
ADC (I2C)	MCP3021A3 T	0x4b	Link
Hőmérő (I2C)	TCN75AVOA	0x48	Link
Karakteres LCD (SPI) + Háttérvilágítás (GPIO)	EA DOGM162W	RS: 27 BL: 18	Link

GPIO

- #include <linux/gpio.h>
- Ellenőrzés

  bool gpio_is_valid(int number);
- Lefoglalás:

  int devm_gpio_request(struct device *dev, unsigned gpio, const char *label);
- Irány:

  int gpio_direction_input(unsigned gpio);

  int gpio_direction_output(unsigned gpio, int value);
- Olvasás

  int gpio_get_value(unsigned gpio);
- Írás:

  void gpio_set_value(unsigned gpio, int value);

Eszközvezérlő regisztrációja

- Az alábbi buszokra regisztrálunk:
  - platform (nem igazi busz)
  - i2c
  - spi

Platform busz

- #include <linux/platform_device.h>
- struct platform_driver
  - int (*probe)(struct platform_device *);
  - int (*remove)(struct platform_device *);
- Regisztráció:
  - module_platform_driver()

I2C busz

- #include <linux/i2c.h>
- struct i2c_driver
  - int (*probe)(struct i2c_client *, const struct i2c_device_id *);
  - int (*remove)(struct i2c_client *);
- Regisztráció:
  - module_i2c_driver()
- Kommunikáció:
  - int i2c_master_send(const struct i2c_client *client, const char *buf, int count);
  - int i2c_master_recv(const struct i2c_client *client, char *buf, int count);

SMBus kommunikáció

- s32 i2c_smbus_read_byte_data(const struct i2c_client *client, u8 command);
- s32 i2c_smbus_write_byte_data(const struct i2c_client *client, u8 command, u8 value);
- s32 i2c_smbus_read_word_data(const struct i2c_client *client, u8 command);
- s32 i2c_smbus_write_word_data(const struct i2c_client *client, u8 command, u16 value);
- Big Endian

SPI busz

- #include <linux/spi/spi.h>
- struct spi_driver
  - int (*probe)(struct spi_device *spi);
  - int (*remove)(struct spi_device *spi);
- Regisztráció:
  - module_spi_driver()
- Kommunikáció:
  - int spi_write(struct spi_device *spi, const void *buf, size_t len);
  - int spi_read(struct spi_device *spi, void *buf, size_t len);

I2C Potméter MCP40D18T

- Cím: 0x2e
- Protokoll:
  - SMBus v2 bájt olvasás és írás
  - Parancskód: 0
  - Érték: alsó 7 bit

I2C ADC MCP3021A3T

- Cím: 0x4b
- Protokoll:
  - Nyers I2C kommunikáció
  - Írás: ping
  - Olvasás: 2 bájt = 4bit semmi, 10 bit info, 2 bit semmi

I2C Hőmérő TCN75AVOA

- Cím: 0x48
- Protokoll:
  - SMBus v2 olvasás és írás
  - A 0-s regiszter olvasásakor: 2 bájt = 1 bit előjel, 7 bit egész érték, 1 bit fél fok, 7 bit semmi

SPI karakteres LCD EA DOGM162W

- Cím: nincs
- Protokoll: csak küldés
  - RS: kontroll / adat
  - Kontroll: külön táblázat tartalmazza
  - Adat: A sor karakterei

Device Tree

- Az I2C, I2S, SPI buszok, illetve a GPIO-k nem teszik lehetővé a detektálást.
- A beágyazott processzorok multifunkcionális lábakkal rendelkeznek.
- Minden HW külön Kernelt igényelt.
- Az egységes Kernelhez a Device Tree leíró állomány szükséges.
- Tartalmazza a használt HW komponenseket a paramétereikkel együtt és a köztük lévő kapcsolatokat.

Device Tree

- A Kernel ez alapján tölti be az eszközmeghajtókat és hívja meg a probe-ot.
- Szöveges formában: *.dts
- Lefordítva: *.dtb
- Az u-boot tölti be a memóriába és adja át a Kernelnek.
- Továbbfejlesztése: Device Tree overlay a bővítő kártyák leírására.

Device Tree overlay - RPi2

- Az …-overlay.dts lefordítása: …-overlay.dtb
- Átmásolás: boot partíció / overlays könyvtár
- A „config.txt” állomány szerkesztése:

  dtoverlay=...
- Reboot