Linux Programozás

Kernel fejlesztés

A Kernel jellemzői

- Monolitikus (egy nagy program az egész)
  - A kezdeti fejlesztése könnyebb volt.
  - Optimalizálható
  - Kevésbé flexibilis
  - Az egyes részek nem fejleszthetőek egymástól függetlenül.
- Betölthető modulok támogatása
  - Pótolja a monolitikus kernelből adódó flexibilitás hiányt.
  - Tárgykódú állomány, amely menet közben linkelődik a kernelhez.
  - A monolitikus kernel miatt erősen verzió függő.

Kernel modulok készítése

A legegyszerűbb kernel modul

- A kernel modulnak minimálisan tartalmaznia kell a következő két függvényt:
  - `init_module()`
  - `cleanup_module()`

Példa: „hellomodule1.c”

Ugyanaz szebben

- Saját init és cleanup függvényneveket is definiálhatunk a következő függvényekkel:
  - `module_init()`
  - `module_exit()`

Példa: „hellomodule2.c”

Fordítás

- A kernel Makefile-jainak felhasználásával történik.
- Hozzunk létre egy Makefile-t a következő sorral:

```
obj-m += hellomodule1.o
```

- Ezt követően használjuk a kernel Makefile-ját úgy, hogy a make parancsot kiegészítjük a következő opcióval:

```
SUBDIRS=<a modul könyvtára>
```

Példa: „Makefile”

Modulok betöltése/eltávolítása

- Betöltés:
  - `insmod <fájl>`
  - `modprobe <modul>`
- Eltávolítás:
  - `rmmod <modul>|<fájl>`
  - `modprobe -r <modul>`

Modprobe

- Kernel modulok egyszerű betöltését teszi lehetővé.
- Egy függőségi fájlt használ, amelyet előtte a depmod készít el.
- Nem csak a kiválasztott modult tölti be, hanem a hiányzó függőségeket is.
- Konfigurációs állománya:
  - `/etc/modprobe.conf`
  - Egyes rendszereknél még: `/etc/modprobe.d`

A modulok és az alkalmazások közötti különbség (1)

- Csak C vagy assembly nyelvet választhatunk.
- A modulok nem szekvenciálisan futnak le, csak hívásokra reagálnak.
- A modulok csak a kernelhez linkelődnek -> nem használhatunk libc függvényeket. Az éppen használható szimbólumok listája: `/proc/kallsyms`
- A linux és az asm könyvtár header állományait használjuk.
- Ha szimbólumokat exportálunk, akkor figyelnünk kell a név megválasztására.

A modulok és az alkalmazások közötti különbség (2)

- Általában a fizikai memóriát használjuk, ezért óvatosan foglaljunk.
- Lebegő pontos számításokat ne használjunk, vagy mentsük az FPU állapotát.
- A kernel modulban elkövetett programozói hibák hatása súlyosabb, a rendszer összeomlásához is vezethet.
- Nincsenek korlátozások, mindenhez jogunk van.
- Konkurencia problémák megszakítások és több processzor esetén jelentkeznek.

A modulok egymásra is épülhetnek

- Ki kell exportálnunk a szimbólumokat, amelyeket a másik modulban használni akarunk.
- Betöltéskor gondoskodnunk kell a függőségekről.
- A modprobe megoldhatja helyettünk. (Ha előtte a „`depmod -a`”-val felderítettük a függőségeket.)

  Példa: „hellomodule3.h” „hellomodule3a1.c” hellomodule3a2.c” „hellomodule3b.c”

- (A példa azt is bemutatja hogyan lehet több állományból felépíteni a modult.)

Export csak GPL modulnak

Az `EXPORT_SYMBOL()` makrónak létezik egy további variánsa, melynek neve `EXPORT_SYMBOL_GPL()`. A működése megegyezik az előzővel, azonban csak GPL licensszel rendelkező modulok számára lesz elérhető a szimbólum.

Paraméter átadás modul számára

- Létre kell hoznunk egy statikus globális változót. Beállíthatjuk az alapértelmezett értékét is.
- A kernel paraméterként való használathoz:
  - Jeleznünk kell, hogy a változó modul paraméter, és meg kell adnunk a változó típusát.
  - `module_param(name, type, perm);`

    `module_param_named(name, variable, type, perm);`
  - `module_param_array(name, type, nump, perm);`

    `module_param_array_named(name, array, type, nump, perm);`
  - `module_param(name, charp, perm);`

    `module_param_string(name, string, len, perm);`

Példa: „helloparam.c”

Paraméterek beállítása

- A paramétereket megadhatjuk a modul betöltésekor parancssori argumentumként.
- A paraméterek leírását a `modinfo` paranccsal nézhetjük meg.
- A paramétereket elérhetjük a `/sys/module/<modul név>/parameters/` könyvtár virtuális állományaival is.
  - A jogosultságok megegyeznek a megadottakkal.
  - Utólagos állításnál konkurencia probléma állhat elő.

Karakteres eszközvezérlő

- Általában ez illeszkedik az egyszerűbb hardver eszközökhöz.
- Az inicializációs és tisztogató függvényekben be kell jegyeznünk, illetve meg kell szüntetnünk az eszközkezelő regisztrációját.
- Az állománykezelő függvények implementációi teszik ki a modul többi részét.
- Az eszközvezérlőt a /dev könyvtárban található eszközállományokkal érhetjük el.

Major és Minor számok

- Az eszköz állományok tartalmaznak egy Major és egy Minor számot.
- A Major szám azonosítja az eszközvezérlőt.
- A Minor számot az eszközvezérlő használhatja amennyiben több állományinterfészt is kezelni akar.

Az eszközállományok dinamikus létrehozása

- A klasszikus a statikus eszközállomány
- Első próbálkozás: devfs
- Jelenleg használt: udev
  - Implementálnunk kell az eszközmodellt

A Linux eszközmodell

Lényegében egy komplex adatstruktúra

- Típus osztály
- Melyik buszra csatlakozik
- Milyen attribútumokkal rendelkezik

Az eszköz létrehozása:

```
struct device* device_create(struct class* osztály,
struct device* szülő, dev_t eszköz, const char*
név, ...);
```

Megsemmisítés:

```
void device_destroy(struct class* osztály, dev_t
eszköz);
```

Eszköz osztály

- Létrehozás:

```
struct class* class_create(struct
module* modul, const char* név);
```

- Megsemmisítés:

```
void class_destroy(struct class*
osztály);
```

Eszközvezérlő regisztrációja

- Karakteres eszközvezérlő regisztrációja:

```
int register_chrdev(unsigned int major,
const char *name, const struct
file_operations *fops);
```

- Ha a major szám 0, akkor dinamikusan allokál egyet.
- A regisztráció megszüntetése:

```
unregister_chrdev(unsigned int major, const
char *name);
```

- A regisztrált eszköz látható a `/proc/devices` állományban.

Állományműveletek

- Az eszközvezérlőnek az alkalmazásokkal való kommunikációhoz az állománykezelő rendszerhívásokat kell implementálnia.
- A `linux/fs.h`-ban található `file_operations` struktúrának megfelelően kell létrehoznunk a függvényeket. (Figyeljünk arra, hogy a struktúra idővel növekszik.)

Adatmozgatás User és Kernel space között

Az eszközvezérlőknek gyakran kell adatot mozgatniuk a user space és kernel space között.

- A `linux/uaccess.h` tartalmaz hozzá függvényeket.

- User space -> Kernel space

```
unsigned long copy_from_user(void *to,
const void __user *from, unsigned long n);
```

- Kernel space -> User space

```
unsigned long copy_to_user(void __user
*to, const void *from, unsigned long n);
```

Egyszerű karakteres eszközvezérlő

Példa: „hellodriver.c”

A Minor szám használata

- Egyik módja, ha a read, write, stb. függvényekben a minor szám alapján létrehozunk egy elágazást.

  Példa: „multi-dev1.c”

- A másik megoldás, ha az open függvényben a minor szám alapján felüldefiniáljuk a file_operations struktúrát.

  Példa: „multi-dev2.c”

A proc állományrendszer használata

Egy csak olvasható proc állomány segítségével egyszerűbben adhatunk információkat a modul állapotáról. Implementálása:

- Létre kell hoznunk egy proc virtuális fájl bejegyzést.
- Be kell állítanunk a file_operations struktúrát.
- A „sequence file” segítségével egyszerű „printf” szerűvé válik az implementáció.

Példa: „helloproc.c”