A PCI Hot Plug eszközillesztő fájlrendszer

Greg ismerteti, hogyan valósít meg a PCI Hot Plug-mag egy RAM-alapú fájlrendszert, és hogyan teheted meg ugyanezt a saját illesztőiddel te is.

Múlt év május 14-én ***H. Peter Anvin*** az alábbiakat tette közzé a „linux-kernel" levelezési listán: „***Linus Torvalds*** ideiglenes szünetet kért az új eszközök azonosítóinak kiosztásában. Reményei szerint ennek köszönhetően egy új és jobb eszközelnevezési rendszer fog kialakulni."
Peter tulajdonképpen a linuxos nevek és azonosítók kiosztásáért felelős „hivatal", így bárkinek, aki rendszermag-illesztőprogram fejlesztésére adja a fejét, hozzá kell fordulnia egy fő- és egy alazonosítóért, amit az illesztőprogramjához kap. Az új azonosítók kiosztásának befagyasztása természetesen parázs vitát gerjesztett: vajon milyen módszerrel lehetne hatékonyabban kezelni az eszközök neveit, azonosítóit? Az egyik ötlet az volt, hogy egy illesztőprogramot kellene készíteni, amely egy a felhasználó és az illesztőprogam közti párbeszédet kezelő fájlrendszert valósítana meg.
Ez idő tájt én éppen a Compaq által a saját kiszolgálóihoz írt PCI Hot Plug illesztőprogramot csinosítgattam. A PCI Hot Plug illesztőprogram lehetővé teszi a PCI-kártyák üzem közben történő kikapcsolását, kivételét a gépből, egy másik kártya behelyezését, majd a foglalat és az új kártya visszakapcsolását – feltéve, hogy megfelelő vassal rendelkezünk. Különösen akkor hasznos az ilyesmi, ha olyan kiszolgálót üzemeltetünk, amelynél a leállás megengedhetetlen, azonban újabb hálózati kártyák telepítésére, a meghibásodottak cseréjére vagy egyéb karbantartási munkák elvégzésére szükség lehet.
A PCI Hot Plug illesztőprogramot eredetileg arra tervezték, hogy karakteres eszközként tartsa a kapcsolatot a felhasználói réteggel. Az `ioctl` hívásokat az eszközcsomóponthoz kellett intézni, így lehetett a PCI-foglalatokat és a foglalatok visszajelző fényeit leállítani és bekapcsolni, valamint az eszközökön különféle hibaellenőrző programokat futtatni. Annak érdekében, hogy a rendszerben lévő PCI-foglalatok számát és állapotát (energiaellátás és állapotjelzők) le lehessen kérdezni, egy csak olvasható */**proc*** könyvtárfa is elérhető volt.
Miközben azon dolgoztam, hogy a PCI Hot Plug központi részeit kiszedjem a Compaq-illesztőprogramból, hogy a felhasználó felé más PCI Hot Plug illesztőprogramok is közös felülettel rendelkezhessenek, rájöttem, hogy egyetlen fájlrendszerrel jobban meg lehetne oldani mind a PCI-foglalatokkal kapcsolatos adatok, mind a felhasználótól érkező beavatkozások kezelését. Az összes, az illesztőprogrammal kapcsolatos adatot és beavatkozási műveletet egyetlen helyről lehetne kezelni, és nem kellene két különböző jellegű felületet fenntartani.
A PCI Hot Plug illesztőprogram a 2.4.15-ös változattól kezdve a fő rendszermagfa részét képezi, és ***pcihpfs*** néven fájlrendszert tesz elérhetővé – az illesztőprogram ennek segítségével kezelhető. Amikor befűzzük a fájlrendszert, egy 3, 4, 5 stb. nevű könyvtárakból álló fát kapunk, melyben a számok a PCI-foglalatok fizikai számát jelzik. Az egyes foglalatokhoz tartozó könyvtárakban található fájlokból olvashatjuk ki a foglalattal kapcsolatos adatokat. A ***power*** és ***attention*** nevű fájlok írhatók is, így `0/1` értéket megadva a tápellátás és a jelzőfény ki- és bekapcsolható. A ***test*** fájlt hardveres tesztparancsok küldésére használhatjuk. Az ***adapter*** állományból tudhatjuk meg, hogy van-e kártya a foglalatban, a ***latch*** fájl pedig a foglalathoz – fizikailag – tartozó retesz állapotáról tájékoztat, feltéve, hogy a gépben található ilyen.
Ha tehát például az 5-ös foglalatot akarjuk bekapcsolni, az alábbi parancsot kell kiadnunk a ***pcihpfs*** főkönyvtárában:

```
echo 1 > 5/power
```

Ha a foglalatban kártya van, a teljes PCI-indítási folyamat le fog futni, amelynek részeként PCI-adatokkal a */**sbin**/**hotplug***-ot is meghívja, így a rendszer az eszközhöz tartozó modult önműködően betöltheti.
A fájlrendszernek köszönhetően a felhasználói programoknak karakteres eszközhöz nem kell különleges `ioctl()` hívásokat intézniük, és számos lehetőség közül választhatnak, hogyan is kívánják eszközeiket vezérelni.
A cikk további részei arról szólnak, hogy a PCI Hot Plug-mag mi módon valósít meg egy RAM-alapú fájlrendszert, és miként teheted meg ugyanezt a saját illesztőiddel te is.
Először is az illesztőprogramban be kell vezetni a fájlrendszert (declaration). Ehhez a `DECLARE_FSTYPE` makrót lehet használni, amely az ***include**/**linux**/**file.h*** állományban található.
A ***pci_hotplug*** illesztőprogram a `DECLARE_FSTYPE` makrót az alábbiak szerint használja:

```
static DECLARE_FSTYPE(pcihpfs_fs_type,
➥"pcihpfs", pcihpfs_read_super,
➥FS_SINGLE | FS_LITTER);
```

Ezzel a `struct pcihpfs_fs_type` típusból egy `file_system_type` nevű statikus példány jön létre, valamint megtörténik a struktúra néhány mezőjének kezdeti értékadása is. A ***name**/**név*** mező értéke most `pcihpfs`, ezt fogják a felhasználók a fájlrendszer befűzésére használni, így nem árt mindig olyan nevet választani, aminek értelme is van, és amelyet más fájlrendszer nem használ a rendszermagban. Az `FS_SINGLE` és `FS_LITTER` jelzőket is állítsuk be.
Az `FS_SINGLE` azt jelenti, hogy fájlrendszerünkben a szuperblokkból csak egyetlen példány lesz. Emiatt bárhova is fűzzük be a fájlrendszert, a fájlrendszerben minden befűzési pont ugyanarra helyre fog mutatni (ne feledd, ugyanazt a fájlrendszert a könyvtárfa különböző pontjain is befűzheted).
Az `FS_LITTER` beállítással megadtuk, hogy a könyvtárfát a `dcache`-ben szeretnénk tartani. A beállítást azért használjuk, mert fájlrendszerünk teljes egészében a memóriában foglal majd helyet, nem lesz lemez, háttértár vagy egyéb fizikai adattároló hozzárendelve.
A `pcihpfs_fs_type` `read_super` mezője arra a függvényre mutat, amelyet akkor hívunk meg, amikor a rendszermag a fájlrendszer szuperblokkját akarja beolvasni. A szuperblokk egy fájlrendszernek az a szerkezete, amely a teljes fájlrendszert magát írja le. A rendszermag ezt a függvényt akkor hívja meg,

amikor valaki be akarja fűzni a fájlrendszert. A függvény meghívásakor a rendszermag tudomására kell hoznunk, hogy a fájlrendszer pontosan hogyan épül fel.
Mielőtt azonban befűzhetnénk a fájlrendszert, meg kell mondanunk a rendszermagnak, hogy maga a fájlrendszer elérhető, létezik. Ezt egy egyszerű `register_filesystem()` hívással tehetjük meg, amelynek egyetlen átadott értéke a `file_system_type`. A műveletet a `pci_hotplug` modul kezdő értékeit megadó függvényével végezhetjük el, az alábbi kódrészlettel:

```
dbg("fÆjlrendszer bejegyzØse.\n");
result = register_filesystem
➥(&pcihpfs_fs_type);
if (result) {
   err("a register_filesystem hibÆval tØrt
   ➥vissza, hibak d: %d\n", result);
  goto exit;
 }
```

A `pci_hotplug` modul leállításakor fájlrendszerünk bejegyzését az alábbi kódrészlettel hasonló módon töröljük:

```
unregister_filesystem(&pcihpfs_fs_type);
```

Miután bejegyeztük a fájlrendszert, néhány virtuális fájlt szeretnénk létrehozni, amelyek révén a felhasználók írhatják és olvashatják az illesztőprogram által módosított és exportált értékeket. Ha egy felhasználó azelőtt fűzi be a fájlrendszert, hogy fájlt próbálna létrehozni, a rendszermag már rendelkezni fog a fájlrendszer egy virtuális megtestesülésével. Előfordul, hogy a fájlrendszer még nincs befűzve, létrehozása után azonban – mielőtt új fájlt hozhatnánk létre – a rendszermagnak be kell fűznie (egyéb esetben az állomány létrehozása sikertelen lesz). Ennek elhárítására két megoldás kínálkozik. Az egyiknél addig várunk, amíg a fájlrendszer befűzése ténylegesen megtörténik (a befűzésről a `read_super` függvény meghívása tudósít bennünket), és csak ezután hozzuk létre az új állományokat. Ennél a módszernél a befűzésnél elég sokat kell dolgoznunk, és folyamatosan figyelemmel kell kísérnünk, hogy a fájlrendszer éppen be van-e fűzve. Ne feledjük ugyanis, hogy különböző időpontokban kell fájlokat hozzáadnunk vagy törölnünk. Az ***usbdevfs*** fájlrendszer (ennek nincs köze a ***devfs***-hez, csak szerencsétlen módon hasonlít a nevük) kiváló példa az olyan fájlrendszerre, amelyik ezt a megoldást alkalmazza.
Most azonban nem kívánjuk folyamatosan követni, hogy fájlrendszerünk be van-e fűzve vagy sem, egyszerűen csak fájlokat akarunk létrehozni és törölni, amikor éppen kedvünk tartja. A második megoldás szerint a rendszermagot arra kell utasítanunk, hogy a fájlrendszert belsőleg fűzze be. Ezzel elhárul a befűzési állapot követésének gondja. Az *1. listán* ez a megoldás látható. Rágjuk át egy kicsit ezt a kódrészletet, és próbáljuk megérteni, mit és hogyan tesz. Arra az esetre nézvést is kiváló példát adunk, ha megfelelő zárolási módszereket kell kidolgozni, mert például a rendszermag többprocesszoros gépen fut.
Az alábbi sorral elsőként egy `spin`-zárat kérünk `mount_lock` néven:

```
spin_lock(&mount_lock);
```

Ezt a zárat belső számlálónk védelmére használjuk abban az esetben, ha a fájlrendszer már be lenne fűzve. Igen, igaz, korábban azt mondtam, hogy a befűzési állapotot nem szeretnénk nyomon követni. Semmi baj nincs, ezzel az egyszerű függvénnyel, valamint a fájlrendszer leválasztására használttal (ezt később ismertetem) sokkal könnyebb dolgozni, és könnyebb őket megérteni is, mint a befűzési állapot folyamatos követésére használt módszert. Elég belenézni a 2.4.18-as és korábbi rendszermagok ***drivers/usb/inode.c*** állományába, és máris láthatjuk, mi kell ez utóbbi megfelelő működéséhez.
Miután a `spin`-zár megvan, ellenőriznünk kell, hogy a belső `mount` változó be van-e állítva:

```
if (pcihpfs_mount) {
        mntget(pcihpfs_mount);
        mntget(pcihpfs_mount);
        spin_unlock (&mount_lock);
        goto go_ahead;
 }
```

Ha igen, az `mntget()` meghívásával növeljük belső befűzési számlálónkat. Az `mntget()` egy egyszerű, helyben kifejtett (inline) függvény az ***include/linux/mount.h*** állományban.

1. lista Kódrészlet, melyben a rendszermagot a fájlrendszer belső befűzésére utasítjuk

```
 static int get_mount (void)
{
    struct vfsmount *mnt;

    spin_lock (&mount_lock);
    if (pcihpfs_mount) {
        mntget(pcihpfs_mount);
        ++pcihpfs_mount_count;
        spin_unlock (&mount_lock);
        goto go_ahead;
    }

    spin_unlock (&mount_lock);
     mnt = kern_mount (&pcihpfs_fs_type);
    if (IS_ERR(mnt)) {
        err ("nem siker lt befßzni a
        ➥fÆjlrendszert  kilØpØs!\n");
        return -ENODEV;
    }
    spin_lock (&mount_lock);
    if (!pcihpfs_mount) {
        pcihpfs_mount = mnt;
        ++pcihpfs_mount_count;
        spin_unlock (&mount_lock);
        goto go_ahead;
    }
    mntget(pcihpfs_mount);
    ++pcihpfs_mount_count;
    spin_unlock (&mount_lock);
    mntput(mnt);

go_ahead:
    dbg("pcihpfs_mount_count = %d\n",
       ➥pcihpfs_mount_count);
    return 0;
}
```

Ezután belső számlálónkat növeljük, eleresztjük a spin-zárat, és a függvény végére ugrunk, ugyanis végeztünk (néha még a rendszermagba is belefér egy-egy goto).
Egyéb esetben a fájlrendszer nincs befűzve. Eleresztjük tehát a spin-zárat:

```
spin_unlock (&mount_lock);
```

majd a kern_mount meghívásával belsőleg fűzzük be a fájlrendszert:

```
mnt = kern_mount (&pcihpfs_fs_type);
if (IS_ERR(mnt)) {
    err ("nem siker lt befßzni a fÆjlrendszert...
        ➥kilØpek!\n");
    return -ENODEV;
 }
```

A spin-zárat eleresztjük, mivel a kern_mount() függvény futása eltarthat egy ideig, és az is előfordulhat, hogy a rendszermag más folyamatnak adja át az erőforrásokat. Ügyeljünk arra, hogy spin-zárat nem szabad fenntartani, ha a schedule() meghívására sor kerülhet – ilyenkor ugyanis elég csúnya dolgok történhetnek…
Most, hogy a fájlrendszert befűztük, újra elkérjük spin-zárat:

```
spin_unlock (&mount_lock);
```

de egyúttal ellenőrizzük, hogy a belső, befűzést jelző számlálónk továbbra is nulla-e:

```
if (!pcihpfs_mount) {
        pcihpfs mount = mnt;
        pcihpfs_mount_count;
        spin_unlock (&mount_lock);
        goto go_ahead;
 }
```

„Állj!" – mondhatnád ekkor. „Mit akarunk a pcihpfs_mount-tal? Tudjuk, hogy nulla az értéke, alig pár kódsorral előbb vizsgáltuk meg! Miért bántjuk újra?" Ne feledd, hogy a kern_mount() meghívásánál – mint szó volt róla – a vezérlés más folyamathoz is kerülhet! Ha a kern_mount() hívásakor ez történik, és egy másik folyamat is ugyanezt a kódrészletet hajtja végre (miért is ne, hiszen több processzorunk van, és egy időben természetesen több felhasználói szál is futhat), előfordulhat, hogy a fájlrendszert sikeresen befűzte, és a pcihpfs_mount változó értékét növelte. Na, ezért kell újra elvégeznünk az értékvizsgálatot.
Ha nem volt másik folyamat, amely közbelépett, és a fájlrendszert befűzte volna, a fájlrendszer mutatóját későbbi műveletekhez mentjük, belső számlálónkat megnöveljük, a zárat eleresztjük és kilépünk.
Ha egy másik folyamat köreinket megzavarta, és befűzte a fájlrendszert, a következőket tesszük:

```
mntget(pcihpfs_mount);
++pcihpfs_mount_count;
spin_unlock (&mount_lock);
mntput(mnt);
```

Ez azzal egyezik meg, amit hasonló helyzetben cselekedtünk a függvény elején.
A fájlrendszer leválasztására használt kódrészlet még egyszerűbb:

```
static void remove_mount (void)
{
        struct vfsmount *mnt;

        spin_lock (&mount_lock);
        mnt = pcihpfs_mount;
        --pcihpfs_mount_count;
        if (!pcihpfs_mount_count)
                ➥pcihpfs_mount = NULL;

        spin_unlock (&mount_lock);
        mntput(mnt);
        dbg("pcihpfs_mount_count = %d\n",
            ➥pcihpfs_mount_count);
 }
```

Ebben a függvényben is szert teszünk egy zárra (a fájlrendszer befűzésekor használttal megegyezőre), a fájlrendszer befűzési alkalmainak számát nyilvántartó számlálót csökkentjük (minden befűzéshez tartoznia kell egy leválasztásnak), majd eleresztjük a zárat. Ezután szólunk a rendszermagnak, hogy a fájlrendszert le akarjuk választani – ehhez már csak az mntput() meghívása szükséges.
Amikor a rendszermag lényegében azért akarja befűzni a fájlrendszert, mert meghívtuk a kern_mount() függvényt – vagy mert egy felhasználó kezdeményezte –, a pcihpfs_read_super() függvény kerül meghívásra. Ebben létre kell hoznunk néhány rendszermagszerkezetet, amelyek megadják a fájlrendszer jellemzőit, valamint a fájlrendszer élettartama alatt a rendszermag által meghívott függvények fellelhetőségét. Mindezt az alábbi néhány kódsorral érhetjük el:

```
sb->s_blocksize = PAGE_CACHE_SIZE;
sb->s_blocksize_bits = PAGE_CACHE_SHIFT;
sb->s_magic = PCIHPFS_MAGIC;
sb->s_op = &pcihpfs_ops;
```

Először is megadjuk, hogy fájlrendszerünk blokkmérete megegyezik a lapozó gyorsítótár méretével, közöljük a fájlrendszer magic-azonosítóját, amelynek a rendszerben egyedinek kell lennie, majd megadjuk super_operations szerkezeti függvényeink listájának elérhetőségét.
A szuperblokk központi fájlleírójának (root inode) kezdeti értékeit az alábbiakkal adjuk meg:

```
inode = pcihpfs_get_inode(sb, S_IFDIR | 0755, 0);
if (!inode) {
       dbg("%s: az i-node nem Ørhetı
       ➥el!\n",__FUNCTION__);
       return NULL;
 }
```

A pcihpfs_get_inode() működéséről még lesz szó, ha viszont a fentiek sikeresek, a fő dentry-t hozzárendeljük a most létrehozott fájlleíróhoz, majd mentjük a dentry-t a szuperblokkszerkezetbe:

```
root = d_alloc_root(inode);
if (!root) {
       dbg("%s: a gy kØr dentry nem Ørhetı
       ➥el!\n",
           __FUNCTION__);
```

2. lista Új fájlleíró létrehozása

```
static struct inode *pcihpfs_get_inode
(struct super_block *sb, int mode, int dev)
{
    struct inode *inode = new_inode(sb);

    if (inode) {
      inode->i_mode = mode;
      inode->i_uid = current->fsuid;
      inode->i_gid = current->fsgid;
      inode->i_blksize = PAGE_CACHE_SIZE;
      inode->i_blocks = 0;
      inode->i_rdev = NODEV;
      inode->i_mapping->a_ops =
      ➥&pcihpfs_aops;
       inode->i_atime = inode->i_mtime =
       ➥inode->i_ctime = CURRENT_TIME;
      switch (mode & S_IFMT) {
      default:
        init_special_inode(inode, mode, dev);
        break;
      case S_IFREG:
        inode->i_fop =
        ➥&default_file_operations;
        break;
      case S_IFDIR:
         inode->i_op =
         ➥&pcihpfs_dir_inode_operations;
         inode->i_fop =
         ➥&pcihpfs_dir_operations;
         break;
      }
    }
    return inode;
}
```

```
      iput(inode);

      return NULL;
}
sb->s_root = root;
return sb;
```

Ez minden, amire a szuperblokk kezdeti értékadásához szükség van – ezzel a rendszermag a fájlrendszert sikeresen befűzte.
A `pcihpfs_get_inode()` is egy olyan függvény, amelyet létre kell hoznunk a fájlrendszerhez. Akkor hívódik meg, amikor a fájlrendszerben új fájlleírót kell létrehoznunk.
A 2. *lista* szemlélteti, miként végzi el mindezt a `pci_hotplug` illesztőprogram.
Először a rendszermag `new_inode()` függvényét kell meghívnunk, hogy új fájlleíró szerkezet jöjjön létre, és a kezdeti értékadás megtörténjen. Ha ez sikeres, több mezőt is fel kell töltenünk a szükséges adatokkal. Az ***i_uid*** és az ***i_gid*** mezőkbe a jelenlegi folyamat `uid`- és `gid`-értékei kerülnek, ily módon biztosítható a fájlleíró létrehozójának későbbi hozzáférése.
Az ***i_atime***, ***i_mtime*** és ***i_ctime*** mezők a fájlleíró hozzáférés idejét, az utolsó módosítás és az utolsó változás idejét adják meg. Ezeket a pillanatnyi időpontra állítjuk be. Ha ez a fájlleíró

5. lista Az fs_remove_file() függvény meghívása

```
static void fs_remove_file
➥(struct dentry *dentry)
{
    struct dentry *parent = dentry->d_parent;

    if (!parent || !parent->d_inode)
         return;

    down(&parent->d_inode->i_sem);
    if (pcihpfs_positive(dentry)) {
        if (dentry->d_inode) {
             if (S_ISDIR(dentry->d_inode->
             ➥i_mode))
                  vfs_rmdir(parent->
                  ➥d_inode,dentry);
             else
                  vfs_unlink(parent->
                  ➥d_inode,dentry);
        }

        dput(dentry);
    }
    up(&parent->d_inode->i_sem);
}
```

„rendes” fájltípus, akkor a `default_file_operations` halmazra mutatunk, mint azon függvények halmazára, amelyeket a fájlleíró szerephez jutásakor – megnyitás, írás, olvasás stb. – kell meghívni. Ha a fájlleíró könyvtárleíró, az alapértelmezett halmaz a könyvtárkezelő függvényekre fog mutatni. Ha a fájlleíró nem „rendes” és nem könyvtárleíró, hagyjuk, hogy a kezdeti értékadást a rendszermag az `init_special_inode()` meghívásával végezze el.
Most, hogy sikeresen elvégeztük a fájlrendszer belső befűzését, hogyan hozhatunk létre a felhasználók által is írható-olvasható fájlokat? Először az `fs_create_file()` függvényt kell meghívni, amelynek átadjuk a létrehozandó fájl nevét és kezelési módját, egy mutatót a szülőkönyvtárra – ha ez `NULL`, a fájl a fájlrendszer főkönyvtárába kerül –, egy másik mutatót a fájlhoz hozzárendelendő adathalmazzal, valamint egy a fájl elérésekor meghívott fájlműveletek halmazát megadó mutatót.
Itt hívjuk meg a `pcihpfs_create_by_name` függvényt, amellyel a megadott adatok alapján egy új `dentry` jön létre.
A `dentry` létrehozása után mentjük az adatmutatót, és a `dentry file_operations` mutatója azokra a függvényekre irányul, amelyeket a `dentry` fájlleírójának elérésekor ténylegesen meg akarunk hívni.
A `file_operations` szerkezet, amelyet egy fájlleíróhoz rendelünk, a létrehozott fájl típusától függően változik. A ***power*** állomány esetében, amely az adott PCI-foglalat ki- vagy bekapcsolt állapotát adja meg, valamint a ki- és bekapcsolásra is használható, az alábbi szerkezetet használjuk:

```
static struct file_operations
power_file_operations = {
        read:           power_read_file,
        write:          power_write_file,
        open:           default_open,
};
```

Ez esetben a `power_read_file` és a `power_write_file` függvény érdekes. Ezek kerülnek meghívásra, ha valaki a fájlt olvasni vagy írni próbálja. A további függvények akkor jutnak szerephez, ha más műveleteket végzünk az állományon. Az `open()` hívás hatására a rendszermag a `default_open`, az `llseek` hatására a `default_file_lseek()` függvényt hívja meg és így tovább.
A `power_read_file()` rendkívül egyszerű függvény – mindössze a megadott PCI-foglalat ki- vagy bekapcsolt állapotát kell visszaadnia. A vonatkozó kódrészlet:

```
page = (unsigned char *)
       ➥__get_free_page(GFP_KERNEL);
if (!page)
          return -ENOMEM;

retval = get_power_status (slot, &value);
if (retval)
         goto exit;
len = sprintf (page, "%d\n", value);
```

A kódrészlet lefoglal egy darab memóriát (egy lapnyit), lekérdezi a PCI-foglalat állapotát (a `get_power_status()` függvény meghívásával), majd az állapotot tükröző karakterláncot kiírja a lefoglalt memóriarészbe. A memóriarészt ezután átmásolja a felhasználói memóriaterületre. Ne feledjük, hogy az eredeti memóriarész a rendszermaghoz tartozik – ha a felhasználók oldaláról is elérhetővé akarjuk tenni, az alábbi kódrészletet végre kell hajtani:

```
if (copy_to_user (buf, page, len)) {
       retval = -EFAULT; goto exit;
 }
```

A részletben a `buf` mutató a felhasználói memóriarészben lévő átmeneti tárra mutat, amelyet eredetileg a `read()` hívásnak adtunk át. Amikor a felhasználó kiadja az alábbi parancsot

```
cat /tmp/pcihpfs/slot2/power
```

az eredmény a következő:

```
1
```

A `power_write_file()` függvény ugyanilyen egyszerű. Azt szeretnénk, ha a felhasználó egy egyszerű `echo` paranccsal vezérelhetné a PCI-foglalat tápellátását, például:

```
echo 1 > /tmp/pcihpfs/slot3/power
```

Ezzel a paranccsal a harmadik PCI-foglalat tápellátását lehet bekapcsolni. A művelethez az átadott értéket karakterlánc formátumból át kell alakítanunk bináris számmá, és meg kell határoznunk, hogy melyik PCI-foglalatokra jellemző egyedi függvényt kell meghívnunk (lásd a 4. listát, 36. CD Magazin/PCI könyvtár).
Először létrehozunk egy a felhasználói karakterláncnál egy bájttal nagyobb átmeneti tárat, és feltöltjük nullákkal. Második lépésként a tartalmát a felhasználói részből a rendszermag átmeneti tárába másoljuk, ezután a `simple_strtoul()` függvénnyel bináris formátumba alakítjuk, majd értékétől függően a `disable_slot()` vagy az `enable_slot()` függvényt hívjuk meg a megadott PCI-foglalathoz.

A fenti két egyszerű függvénnyel bármely felhasználó által elérhető illesztőprogram-felületet hozunk létre anélkül, hogy különleges, `ioctl`-jellegű hívásokat kellene végrehajtani.
Amikor az illesztőprogram leáll, minden olyan fájlt el kell távolítania, amelyet a fájlrendszerben hoztunk létre, így válik lehetővé a fájlrendszer leválasztása és a hozzárendelt memória felszabadítása. Ehhez az `fs_remove_file()` függvény kell meghívni (lásd az *5. listát*).
A függvénynek át kell adni egy mutatót, amely az `fs_create_file` hívás által visszaadott `dentry`-t határozza meg. Megvizsgálja, hogy a `dentry` rendelkezik-e érvényes szülővel, ugyanis egy `dentry` csak ebben az esetben távolítható el. Ezt követően a rendszermag VFS-rétegétől kéri a `dentry` eltávolítását (ekkor eltérő hívásokra kerülhet sor, attól függően, hogy a `dentry` fájlra vagy könyvtárra hivatkozik).
Leírtuk az alapvető fájlrendszerműveleteket, amelyek egy illesztőprogramban megvalósított fájlrendszer létrehozásához szükségesek. Ha pontosabb leírást szeretnél arról, hogy a különféle részek hogyan működnek együtt, vess egy pillantást a Linux-rendszermagfa ***drivers/hotplug/pci_hotplug_core.c*** állományában található programkódra.
A cikk a 2.4-es rendszermagnál szükséges tennivalókat tárgyalta. A 2.5-ös változatnál – annak köszönhetően, hogy a `ramfs` függvények túlnyomó részét exportálják – számos dolgot könnyebben meg lehet oldani. A RAM-alapú fájlrendszerek között így nagyobb mértékben oszthatók meg a programkódok, a programozók tehát kevesebb munkával is célt érhetnek, és a hibás megvalósítások készítésének valószínűsége is csökken.

Köszönetnyilvánítás

Szeretnék köszönetet mondani ***Pat Mochelnek*** a ***ddfs/driverfs*** kód megírásáért, amelyen a ***pcihpfs*** programkód jelentős része alapult. A ***driverfs*** egy új fájlrendszer a 2.5-ös rendszermagban, amelynek segítségével az illesztőprogramok készítői illesztőprogramjaik különleges adatait a felhasználói területre exportálhatják, valamint faszerkezetben az összes eszközt elérhetővé tehetik, így a tápellátást kezelő eszközök sokkal egyszerűbbekké válhatnak.
Szintén szeretném megköszönni ***Al Viro*** értékes válaszait a VFS-sel kapcsolatos kérdéseimre, amelyek révén egy fájlrendszert ilyen kevés kóddal sikerült megvalósítani.

A listák megtalálhatóak a 36. CD Magazin/PCI könyvtárában.

Linux Journal 2002. május, 97. szám

Greg Kroah-Hartman
(greg@kroah.com) jelenleg a Linux USB és a PCI Hot Plug rendszermagfelelőse. Az IBM-nél dolgozik, ahol számos, a Linux rendszermagjával kapcsolatos kérdéssel foglalkozik.

További érdekességek

Ha további tájékoztatást szeretnél kapni a PCI Hot Plug illesztőprogrammal kapcsolatban, látogass el a ➲ http://www.kroah.com/linux/hotplug címre, ahol a *pcihpfs* kezelését grafikus felületen lehetővé tévő felhasználói alkalmazásokra is találsz hivatkozásokat.