Halmozható fájlrendszerek készítése

Mostantól újraírás nélkül is új képességekkel bővíthetjük kedvenc fájlrendszerünket.

Fájlrendszert, de egyáltalán bármilyen rendszermagkódot írni meglehetősen nehéz feladat. A rendszermag összetett környezet, ahol egyetlen apró hiba is végzetes adatvesztéshez vezethet. A fájlrendszerek a felhasználói alkalmazások szemszögéből nézve átlátszó, egyszerű adatelérési felületet kínálnak, így a fejlesztők állandóan új képességekkel szeretnék bővíteni a fájlrendszereket. E cikkben gyors útmutatást adunk, ami alapján bárki anélkül adhat hozzá új képességeket meglévő fájlrendszeréhez, hogy rendszermag vagy fájlrendszertudor lenne.

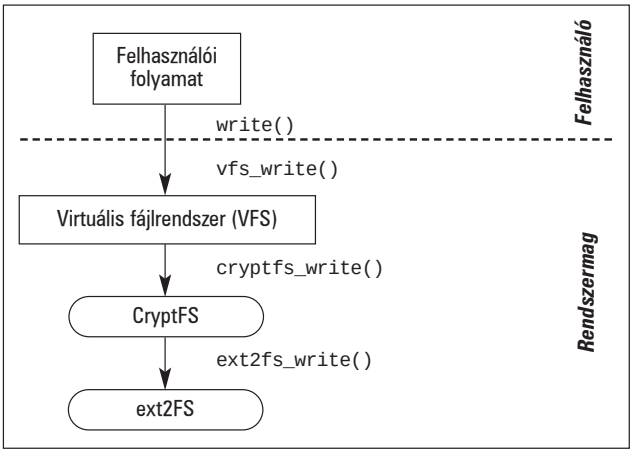
Szóval szeretnél fájlrendszerfejlesztő lenni?

A Linux több hasonló fájlrendszertípust is támogat: lemezes fájlrendszerek, hálózati fájlrendszerek stb. Egy hatékony és megbízható fájlrendszer kifejlesztése éveket vesz igénybe, és ha egyszer végre üzembiztosan fut, nem szívesen tennénk tönkre új szolgáltatások beiktatásával. Ezenkívül a fájlrendszerek karbantartói igen ritkán fogadnak el új képességeket vagy fejlesztési foltokat üzembiztos fájlrendszerükhöz. Így aztán egyáltalán nem meglepő, hogy a jelenleg használatos legnépszerűbb fájlrendszerek évek óta szinte semmit nem változtak. Tegyük fel, hogy egy egyszerű titkosító fájlrendszert szeretnénk készíteni, ami valamilyen előre megadott kulccsal titkosítja az adatot. A különféle kódolásokhoz könnyedén megszerezhetjük a hordozható C-kódot. Ezután a rendszerbe kell illesztenünk az adatvermet kódoló és dekódoló hívásokat. Elméletileg a feladat egyszerű: az írási rendszerhívásokból érkező összes adatot – mielőtt a lemezre írnánk – titkosítjuk, a lemezről érkező adatokat pedig visszafejtjük, még mielőtt a rendszerhívást kiadó folyamatnak visszaadnánk.

Ha módosításról van szó, az ember első gondolata az, hogy lemásolja az `ext2` 5000+ soros forráskódját, áttanulmányozza, majd beilleszti a kódolórészeket. De inkább álljunk ellen a késztetésnek, és ne másoljunk le egy egész fájlrendszert. Igaz ugyan, hogy mindössze 5000+ sorból áll, de a rendszermagkódot legalább egy nagyságrenddel bonyolultabb fejleszteni, mint a felhasználói szintű programokat. Mégha végül sikerül is kódoló eljárásainkat beilleszteni a megfelelő helyre, rá kell ébrednünk, hogy időnk nagy részét tanulmányozással töltöttük, s végül csak néhány helyen szúrtunk be pár sornyi kódot. És mit kaptunk eredményül? Egyetlen, titkosítással kibővített `ext2` fájlrendszert. Mi történik, ha a titkosítót inkább NFS fájlrendszerrel vagy valamely másik rendszer alatt szeretnénk használni a számtalan Linux-fájlrendszer közül?

Egymásra épülő fájlrendszerfejlesztés

A Linux, akárcsak a legtöbb operációs rendszer, két részre bontja a fájlrendszerkódot: vannak belső részei (`ext2`, NFS stb.), és van egy általános célú rétege, amit virtuális fájlrendszernek neveznek (VFS). A VFS réteg a rendszerhívások és a belső fájlrendszerek között helyezkedik el. A VFS célja egységes elérést biztosítani a különböző fájlrendszerekhez, elrejtve azok pontos részleteit. Amikor a fájlrendszerek betöltődnek a rendszermagba, beállítanak néhány függvénymutatót (OO-szóhasználat szerint metódust), amiket azután a VFS használhat. A VFS általában ezeket a függvényeket hívogatja, mit sem sejtve arról, hogy pontosan milyen fájlrendszerek függvényei rejlenek a mutatók mögött. Például az `unlink` rendszerhívás a `sys_unlink` szolgáltatáshívássá alakul, és meghívja a `vfs_unlink` VFS-függvényt, ez azután a feltelepített mutató alapján elindítja a fájlrendszerfüggő függvénymutatót: `ext2` esetében ez az `ext2_unlink` lesz, NFS esetében az `nfs_unlink`, más fájlrendszerek esetében pedig a számukra megfelelő függvényt. Ebben a cikkben a fájlrendszer függvénymutatókat a `->` jellel fogjuk jelölni, a következő formában: `->unlink()`.



1. ábra Példa a halmozható titkosító fájlrendszerre

Mivel mi a titkosító fájlrendszerünket gyorsan szeretnénk fejleszteni, a következő szállóigét alkalmazzuk: „A számítástechnikában egy újabb közvetítő szint felvételével minden feladat megoldható.” Szerencsére a Linux VFS lehetővé teszi, hogy a VFS és egy másik fájlrendszer közé újabb fájlrendszert szúrjunk be. Az *1. ábra* egy ilyen, CryptFS nevű halmozható (stackable) titkosító fájlrendszert mutat be. A CryptFS-t azért nevezzük halmozhatónak, mert egy másik fájlrendszer (`ext2`) tetejére pakoljuk rá. A VFS réteg a CryptFS `->write()` függvénymutatóját (`cryptfs_write`) hívja meg; a CryptFS titkosítja a felhasználói adatokat, fogadja, majd továbbadja az alatta elhelyezkedő `->write()` függvénymutatónak (`ext2_write`). A halmozható fájlrendszerek általában önmagukban is állhatnak, és bármilyen már létező fájlrendszert hordozó befűzési pont fölé felrakhatók; ez annyit jelent, hogy (halmozható) fájlrendszerünket csak egyszer kell kifejlesztenünk, és az valamennyi helyi (alacsonyszintű) fájlrendszeren (`ext2`, NFS stb.) működni fog. Továbbá a Linux 2.4.20-astól kezdve, a halmozható fájlrendszerek távoli NFS-ügyfelekre is biztonságosan átvihetők (az `nfs-utils-1.0` vagy újabb változat segítségével).

Hogyan működik a halmozható fájlrendszer?

A halmozható fájlrendszer alapszolgáltatása műveletet és értékeket átadni a lejjebb található fájlrendszernek. Az alábbi

kódkivonatban bemutatjuk, hogyan kezeli a WrapFS nevű, egyszerű, valóságos feladatnélküli adattovábbító fájlrendszer az `->unlink()` műveletet:

```
int wrapfs_unlink(struct inode *dir,
                  struct dentry *dentry)
{
  int err = 0;
  struct inode *lower_dir;
  struct dentry *lower_dentry;

  lower_dir = get_lower_inode(dir);
  lower_dentry = get_lower_dentry(dentry);

  /* pre-call code can go here */
  err = lower_dir->i_op->unlink(lower_dir,
                        ➥lower_dentry);
  /* post-call code can go here */

  return err;
}
```

Amikor a VFS szeretne letörölni egy fájlt a WrapFS fájlrendszerről, meghívja a `wrapfs_unlink` függvényt, és átadja a törlendő fájlt tartalmazó könyvtár fájlleíró (inode) adatát (`dir`), valamint a törlendő bejegyzés nevét (a dentry belsejében található).
Minden egyes fájlrendszer nyilvántart néhány szükséges objektumot, például fájlleíró-azonosítókat, könyvtárbejegyzéseket, könyvtárneveket és nyitott fájlokat. Amikor halmozást alkalmazunk, akár több objektum is jelölheti ugyanazt az állományt – csak éppen különböző rétegeken. Például a ábrán látható CryptFS lehet, hogy fenntart egy könyvtárbejegyzés-(dentry) objektumot a fájlnév nyílt szöveges változatával feltöltve, míg az `ext2` ugyanennek a névnek a kódolt változatát tárolja egy másik könyvtárbejegyzésben. Amennyiben a halmozható fájlrendszer igazán átlátszó akar maradni a VFS és a többi fájlrendszer számára, kénytelen minden szinten több objektumot is fenntartani.
Ez az oka annak, hogy a `wrapfs_unlink` első dolga a megkapott értékek alapján megkeresni az adott objektumhoz tartozó könyvtárbejegyzés fájlleíróját az egy szinttel lejjebb befűzött fájlrendszeren. Ezek a `get_lower_*` függvények lényegében a WrapFS objektumok (létrehozásakor kitöltött) saját mezőiben található, előre tárolt mutatókat követik. Az alsóbb objektumok beazonosítását követően megkezdődhet a halmozás lényegi része. Meghívjuk az alacsonyabb szinten található fájlrendszer `->unlink()` függvénymutatóját az alacsonyabb szinten elhelyezkedő könyvtár fájlleírója alapján, és átadjuk a két alacsonyabb szintű objektumot.
A WrapFS teljes értékű nullréteg (vagy loopback) fájlrendszer, ami egyszerűen (és minden módosítás nélkül) csak műveleteket és objektumokat közvetít a VFS és az alul elhelyezkedő fájlrendszer között. Sajnos azért mégsem olyan könnyű dolog megírni a WrapFS-t, ugyanis az alsó fájlrendszerrel szemben VFS-ként kell viselkednie, ugyanakkor a Linux igazi VFS-rétege számára alsóbb fájlrendszernek kell látszania. Ez a kettős szerep a kulcsok, hivatkozásszámlálók és a memóriafoglalás óvatos használatát követeli meg. Szerencsére valaki volt olyan jó, és előre elkészítette, valamint fenntartja nekünk a WrapFS-réteget. A WrapFS-réteget nagyszerűen fel tudjuk használni mintaként, ha módosításokat vagy új szolgáltatásokat szeretnénk megvalósítani.

Nekirugaszkodás

Most már világos, hogyan működik a halmozás, de mi a következő lépés? Először is keressük meg azokat a helyeket a WrapFS-ben, ahová a saját kódunkat beilleszthetjük. Az imént bemutatott `wrapfs_unlink` kódhoz visszatérve három ilyen helyet találunk, az alsóbb szintű `->unlink()` függvénymutató előtt, után vagy helyett.

1. Előhívás: kódunkat beilleszthetjük az alacsonyabb szintű `->unlink()` hívás elé. Például ellenőrizhetjük, hogy a felhasználó fontos fájlt akar-e törölni, és ha igen, megakadályozhatjuk:
```
if (strcmp(dentry->d_name.name,
           "vmlinuz") == 0)
return -EACCES;
```
2. Hívás: az egész hívást is helyettesíthetjük. Például a fájl törlése helyett esetleg csak át akarjuk nevezni valamilyen egyszerű kis visszafordítható (undo) fájlrendszer részeként (valószínűleg mindannyian találkoztunk már a véletlenül kiadott `rm -f` parancsok hatásával).
3. Hívás után: itt az alsóbb fájlrendszerből visszatérő főművelet után tudunk egyéb műveleteket végrehajtani. Tegyük fel, hogy egy rossz szándékú felhasználó törölni akarja a */etc/passwd* fájlt, de a Unix jogosultsági rendszere ezt megakadályozza. A rendszergazda naplózni szeretné (a `syslogd` segítségével) az ilyesfajta eseményeket:
```
if (err == -EACCES &&
    strcmp(dentry->d_name.name,
           "passwd") == 0)
  printk("uid %d tried to delete passwd",
      current->fsuid);
```

A `current` mindig az éppen futó folyamatra mutató általános hatókörű (global) változó, az `->fsuid` pedig e folyamat azonosítója, amit a fájlrendszerek felhasználhatnak.
A fenti és az ezután következő példákat némileg leegyszerűsítettük, hogy a lényegi részek jobban láthatók legyenek, és kevesebb helyet foglaljanak. Például a `d_name.name` összetevő nem null végződésű (null terminated), így megfelelő hosszúságú `memcmp` utasítást kellene használni; illetve annak ellenőrzésére, hogy a dentry által megjelölt fájl valóban a */etc/passwd* állomány, meg kellene néznünk, hogy valóban ez a fájlrendszer-e a gyökérrendszer, vagy a `d_path()` segítségével meg kellene vizsgálnunk az abszolút elérési utat.
A 2.4.20 alatt kipróbált teljes példákat a FiST-honlapon, illetve a 48-as CD Magazin/FiST könyvtárában találjuk meg (➲ http://www.cs.sunysb.edu/~ezk/research/fist).

Ki felügyeli a kukkolókat?

A Unix megpróbálja védelmezni a fájlokat a jogosulatlan felhasználóktól. Amikor a felhasználó olyan fájlt akar megnyitni, amire nincs jogosultsága, a Unix azon nyomban egy „jogosultság megtagadva” hibaüzenetet ad vissza. Néhány felhasználó szeret mások állományai között nézelődni, néha kifejezetten olyan fájlokat keresve, amik véletlenül maradtak védtelenek, vagy megpróbálja kitalálni az esetleg létező fájlneveket valamelyik nem kereshető könyvtárban. Sajnos, mégha ezek a kukkolók nem is járnak sikerrel, az áldozatok többnyire nem sejtik, hogy ilyen jellegű próbálkozás történt.
Az egyik leggyakrabban használt fájlrendszerművelet a `->lookup()`, ami a rendszer minden egyes fájlnévhasználata során meghívódik. A rendszermagnak át kell alakítani a nevet (a karaktersorozatot) a tényleges VFS-objektum azonosítóne-

vére, vagyis inode, dentry vagy fájl alakba. Hogy kiszűrhessük a kukkoló felhasználókat, a következő kódot illesztettük a `snoopfs_lookup` vagy `snoopfs_permission` eljárásokba, közvetlenül az alsó fájlrendszer `->lookup()` hívása után:

```
if ((err == -EACCES ||
     err == -ENOENT) &&
    dir->i_uid != current->fsuid &&
    current->fsuid != 0)
  printk("snoop uid=%d pid=%d file=%s",
         current->fsuid, current->pid,
         dentry->d_name.name);
```

Megvizsgáljuk az alsóbb fájlrendszer `->lookup()` függvényétől visszakapott hibakódot (`err`). Amennyiben EACCES (engedély megtagadva) vagy ENOENT (nincs ilyen nevű fájl vagy könyvtár) állapotú és a könyvtár tulajdonosa (`dir->i_uid`) különbözik a pillanatnyi folyamatot futtató felhasználótól (`current ->fsuid`), de a jelenlegi felhasználó nem a rendszergazda (hiszen a rendszergazda bármit megtehet), akkor üzenetet jelenít meg, amiben megjelöli a kukkoló felhasználó azonosítóját. Az ilyen üzenetek általában a `syslogd`-be íródnak.

Titkosítás

Az eltérítő (wrapper) módszerek különösen alkalmasak a biztonsággal kapcsolatos alkalmazások fejlesztésére, ahol gyakran sokat segíthet az eltérítés vagy a megfigyelés. Nem meglepő, hogy a FiST legnépszerűbb alkalmazásai éppen a titkosított fájlrendszerek. Ebben a példában egy egyszerű, ***rot13*** kódolást alkalmazó titkosító fájlrendszert mutatunk be.
Rendszerünkön minden adatot a (feltételezhetően már korábban elkészült) ***rot13*** nevű algoritmussal szeretnénk titkosítani, ami értékként a bemenő és kimenő verem címét, valamint azok hosszát kapja meg. A korábbi példáinkkal ellentétben azonban most nincsen olyan nevezetes függvénymutató, ahová a `rot13()` függvényünket egyszerűen behelyezve elkódolhatnánk a fájlokat. Valójában bármely fájlrendszeren legyünk is, a fájlok adataival dolgozni meglehetősen összetett dolog, hiszen több függvénymutatóra is ügyelnünk kell, valamint a fájl adatai kétféleképpen, olvasási és írási rendszerhívásokkal is elérhetők, továbbá mindennek bármilyen fájleltolással, valamint az egész lapokon dolgozó `mmap`-pel is együtt kell működnie. A WrapFS, hogy megkönnyítse a halmozható fájlrendszer-fejlesztők életét, a fent felsorolt valamennyi függvénymutatót két egyszerű függvényben foglalja össze: az egyikben kódoljuk a fájladatokat, a másikban visszafejtjük őket, feltételezve, hogy egész laphatáron kezdődő adatokat használnak (például az IA-32 rendszereken ez 4 KB-ot jelent). A WrapFS-mintát használva az egyetlen dolog, amit nekünk kell megírnunk, a ***rot13*** alapú titkosító fájlrendszer. (A példa a 48. CD Magazin/FiST/1lista.txt-ben található.)
A WrapFS már eleve tartalmazza a kevert olvasásokat, írásokat és a memóriatérképes műveleteket kezelni képes összes bonyolult kódot. A WrapFS az `encode_block` segítségével kódolja az adatlapokat, illetve a `decode_block` alkalmazásával visszakódolja őket (példánkban e kettő azonos).
A ***rot13*** nem éppen legcélszerűbb kódolás, de az egyszerű példa alapján sokkal erősebb titkosítást használó fájlrendszereket is építhetünk. Ezt követtük mi is, amikor nemrégiben felépítettünk egy NCryptFS nevű, igen hatékony titkosító fájlrendszert (a CryptFS utódját). Az NCryptFS többféle kódolást is ismer; egy felhasználó, folyamat vagy csoport több kulcsot is használhat; többféle azonosítási módszere van; a kulcsok ideje lejárhat és visszavonhatók; kezeli az átruházott előjogokat; és még sok más képességgel is rendelkezik – mindezt elhanyagolható teljesítménycsökkenés árán.
A WrapFS a fájlnevek módosítását is kezeli két további eljárással, amik a fájlneveket kódolják és dekódolják. Mindenképpen ügyelnünk kell arra a fájlnevek kódolásakor, hogy a fájlneveknek a kódolás után is szabályosak maradjanak. Más szavakkal, nem tartalmazhatnak `null` értékeket vagy / (perjel) karaktert. Általánosan elterjedt megoldás, hogy titkosítás után `uuencode`-oljuk a fájlneveket.

Új képességek felhasználói alkalmazásokhoz

A `wrapfs_unlink` példánkban azt javasoltuk, hogy a fájlt törlése helyett egyszerűen csak nevezzük át, így mentve minden törölt állomány egy másolatát. Tegyük fel, hogy rendszerünket unrmFS-nek nevezik, ahol a törlendő fájlokat törlés helyett inkább eredeti F nevükről ***F.unrm*** alakúra nevezzük át. Elég bosszantó lenne, ha az összes ***.unrm fájl*** megjelenne a könyvtárunkban, főként ha ott egyáltalán nem is számítunk fájlokra. Továbbá ezekkel a szolgáltatásokkal akár a támadót is becsaphatjuk, aki megpróbálja letörölni a naplófájlokat, hogy eltüntesse a nyomait. Mindehhez azonban a ***.unrm*** fájloknak alapértelmezés szerint láthatatlanoknak és elérhetetlennek kell lenniük a felhasználó számára.
Ha bizonyos fájlokat el akarunk rejteni fájlrendszeren, két dolgot kell megtennünk. Először is meg kell akadályoznunk, hogy a fájl megjelenjen a `->readdir()` hívásokban. Ezt úgy tehetjük meg, hogy a `wrapfs_filldir` függvénybe olyan kódot illesztünk, ami az összes `->filldir()` híváshoz kerülő fájlnevet ellenőrzi, és `NULL` értéket ad vissza azokra a fájlokra, amiket nem szeretnénk megjeleníteni. Másodszor, meg kell akadályoznunk, hogy a felhasználók közvetlenül rákeressenek az állományra. Ezt úgy érhetjük el, hogy a `wrapfs_lookup` elején kikeressük a ***.unrm*** fájlokat.
Természetesen, ha ezeket a fájlokat mindenki elől elrejtjük, abból sok hasznunk nem származik. A jogosult felhasználóknak – bizonyos körülmények között – el kell tudniuk érni a fájlokat. Egyszerű megoldás például, ha a hívófolyamat UID-jét ellenőrizzük, és csak bizonyos felhasználók elől rejtjük el a ***.unrm*** állományokat. Ennél jobb megoldás, ha minden rendszerhívások anyját, az `ioctl` hívást alkalmazzuk. A WrapFS rendszerhez tetszés szerinti számban határozhatunk meg új `ioctl` hívásokat, hogy azután apró felhasználói programokkal kiaknázhassuk őket. Ezt a módszert használtuk például a titkosító fájlrendszerünkben, ahol a felhasználó titkosító kódját egy felhasználói szintű eszköz adja át a rendszermagnak.
Az unrmFS rendszerünkben tehát készíthetünk egy visszaállító `ioctl` hívást, ami a visszaállítandó F fájlnevet várja bemenetként, és ellenőrzi, hogy létezik-e ***F.unrm*** nevű állomány. Amennyiben igen, az ***F.unrm*** fájlt visszanevezi F-re, ismét megjelenítve az unrmFS-ben. (A példa a 48. CD Magazin/FiST/2lista.txt-ben található.)
A FiST programot, a leírást és a rengeteg további példát a ➲ http://www.cs.sunysb.edu/~ezk/research/fist címen találjuk.
Jó halmozást!

Linux Journal 2003. május, 109. szám

Erez Zadok (ezk@cs.stonybrook.edu)
A Stony Brook Egyetem Számítástechnika karán dolgozik, a Linux NFS and Automounter Administration szerzője, a FiST halmozható mintarendszer megalkotója, és az Am-utils önműködő befűzőrendszer első számú karbantartója.