A naplózott fájlrendszer, a biztonság és az ACL (15. rész)

A naplózott fájlrendszer után azokat a védelmi mechanizmusokat mutatjuk be, amelyek segítségével adatainkat megvédhetjük a rendszer többi használójától.

Az előző részben a „hagyományos” fájlrendszerekkel foglalkoztunk, amelyek jól működnek ugyan, de korszerűeknek semmiképp sem nevezhetők. Ennek oka a technológiai fejlődés érdekes aszimmetriája. Míg a processzorok sebessége nagyjából másfél évente megkétszereződik, a memóriák mérete pedig az idő előrehaladtával exponenciálisan növekszik (ez az úgynevezett Moore-szabály, ami 1965 óta nem vesztette érvényességét), addig a merevlemezek sebessége (pontosabban a fej pozícionálásának ideje) alig-alig változik.
A memóriák árának csökkenése és méretük gyors ütemben történő növekedése a gyorsítótárak növekedésével is együttjár. Ha a gyorsítótár mérete nő, több blokk fér bele, így több olvasási műveletet tudunk a lemezhez való fordulás nélkül kiszolgálni. Már a 90-es évek elején felmerült, hogy a jövőben az előreolvasási módszer (a blokkokat még azelőtt beolvassuk a gyorsítótárba, mielőtt valóban szükség lenne rájuk) a jövőben nem fog jelentős teljesítménybeli növekedéssel járni. Ebből adódóan a lemezhez való fordulások többsége írási művelet lesz.
Az igazi gond abból adódik, hogy a legtöbb rendszerben az írás mindig csak kis részekben zajlik, és ezek a kis részek általában egymástól távol helyezkednek el. Ez pedig sok pozícionálással, illetve a lemez sokszori forgatásával jár. Természetesen „gyorsítótárazhatnánk” itt is, de a legtöbb esetben (például új állomány létrehozásakor) a fájlleírókat (inode, illetve nem Unix-rendszerekben olyan blokkokat, amelyek a fájlrendszer szempontjából alapvetők) is írnunk kell a lemezre. Az ilyen írásokat azonban célszerű azonnal elvégezni, különben elég egy áramszünet, és máris felborul fájlrendszerünk egységessége.

Naplószerkezetű fájlrendszer

Szükség volt hát egy olyan fájlrendszerre, ami a mai körülményekhez edződött, azaz a „hagyományos” fájlrendszerekkel szemben sok kis írásnál sem veszít a hatékonyságából. Ez az úgynevezett naplószerkezetű fájlrendszer, az LFS (Log-structured File System). A különbség azonban nem a fájlok és a könyvtárak megvalósításában mutatkozik – itt is megtalálhatjuk a fájlleírókat és a fájlok tartalmának elérésében sincs komolyabb változás –, az eltérés a blokkok elhelyezkedésében rejlik.
A „hagyományos” fájlrendszerek esetében megszokhattuk, hogy a fájlrendszer szerkezetét leíró blokkok (Unix esetében ezek a fájlleírók) mindig egy kitüntetett helyen találhatóak a lemezen. Az LFS szakít ezzel a „hagyománnyal”. A lemezen nem azt tárolja, hogy mik találhatók az egyes állományokban, hanem azt, hogy milyen műveletek történtek a fájlrendszerrel. A lemezt tehát egy naplónak tekinti, amiben feljegyzi a fájlrendszerben történt változásokat.
Az LFS-t nem hiába nevezik naplószerkezetűnek, mivel a működése tényleg egy hétköznapi napló írásával egyezik meg. A napló általában egy könyv vagy egy füzet, amit az emberek arra használnak, hogy beleírják a nap történéseit.
A naplókat általában az elejéről kezdik el írni, és minden oldalra egy-egy nap története kerül.
Az LFS ugyanezt teszi: a memóriában összegyűjti a fájlrendszerben történő változásokat. Ha például megváltoztatjuk egy blokk vagy egy fájlleíró tartalmát, akkor az úgy, ahogy van, bekerül ebbe a memóriarészbe. Ezeket a függőben lévő írásokat úgynevezett bejegyzésbe pakolja, ezt pedig egy az egyben kiírhatjuk a lemezre, azaz hozzáfűzhetjük a naplónkhoz.
Minden bejegyzés mérete kötött, általában 1 MB. Az, hogy egy bejegyzés mit tartalmaz, attól függ, hogy milyen műveletek történtek. Véletlenszerűen találhatók benne adatblokkok és fájlleírók is. Minden bejegyzés elejére kötelező azonban egy összefoglalót tenni, ami elárulja nekünk, mi is az, amit az adott bejegyzés tartalmaz. Ezt úgy képzelhetjük el, mintha valaki úgy írná a naplóját, hogy először címszavakban összefoglalja a nap legfontosabb eseményeit, majd részletesen kifejti azokat.
Mivel a naplót lineárisan írjuk a lemezre, a lehető leghatékonyabban tudjuk az írási műveleteket elvégezni. A fájlleírók azonban a lemezen össze-vissza, a napló különböző bejegyzéseiben helyezkednek el, így a fellelésük sokkal bonyolultabb, mint a „hagyományos” fájlrendszerek esetében – az LFS ezért egy táblázatnak is helyet szorít a lemezen, ami az összes fájlleíró pontos helyét megadja.
A naplónknak otthont adó füzet előbb-utóbb betelik. Ilyenkor érdemes elbaktatni a legközelebbi papírboltba, és venni egy újabb füzetet. Az LFS esetében a helyzet összetettebb, hiszen a felhasználók fenntartással kezelnek egy olyan rendszert, ami időnként új merevlemez beszerzésére szólítaná fel őket.
Ám míg a hétköznapi naplók bejegyzései bármikor hasznunkra lehetnek, az LFS régebbi naplóbejegyzései sok felesleges dolgokat tartalmazhatnak, ilyenek az azóta már letörölt vagy felülírt állományok adatblokkjai.
Ennek következtében a háttérben mindig futnia kell egy úgynevezett takarító folyamatnak, aminek az a dolga, hogy ezeket a felesleges adatokat kiradírozza a naplóból, helyet szabadítva így fel az újabb naplóbejegyzéseknek. A gyakorlatban ez úgy működik, hogy a takarító a napló elejéről (tehát az első bejegyzéstől) elindul, s azokat az adatokat gyűjti össze egy újabb bejegyzésbe, amelyek még mindig érvényesek, ezután kiírja őket a napló végére. Ezt követően ezeket a bejegyzéseket felszabadítja, tehát ha betelt a merevlemez, az LFS ismét igénybe veheti őket. (Ha nagyon menőnek szeretnénk tűnni, akkor úgy kellene fogalmaznunk, hogy az LFS a lemezt egyfajta cirkuláris pufferként használja: az új bejegyzések a napló végére kerülnek, míg a takarító folyamat a napló elejéről csíp le egy-egy bejegyzést. A dolog azért ennél sokkal körmönfontabb, a fájlleírók helye ugyanis a takarítás közben megváltozik, így frissíteni kell egyrészt a fájlleírók helyét megadó táblázatot; másrészt – ha a kérdéses fájlleíró közvetett volt, lásd sorozatunk előző részében, akkor – más csomópontokat is.)
Az LFS tehát hatékonyabban kezeli a sok kis véletlenszerű írást, mint a „hagyományos” fájlrendszerek, ennek

azonban megvan az ára: bonyolódik a fájlok tartalmának elérése, illetve a takarításkor rendkívül sok felügyeleti tevékenység elvégzésére kényszerülünk. Emiatt felvetődhet a kérdés, hogy mindent egybevéve az LFS hatékonyabb-e, mint a többi fájlrendszer. A mérések azt mutatják, hogy ennek ellenére is érdemes használni az LFS-t. Hiszen a processzor és a memória ma már sokkalta gyorsabb, mint régen, így hiába több a munka, mégis nagyobb teljesítményt érhetünk el az LFS-sel, ha sok kis véletlenszerű írási műveletnek kell eleget tennünk. Mi a helyzet azonban a nagyméretű írásoknál, illetve az olvasásnál? A mérések szerint semmivel sem marad el a „hagyományos" fájlrendszerektől. Fontos, hogy semmiképp se keverjük össze az LFS-t az úgynevezett naplózó (journaling) fájlrendszerekkel – ez két teljesen különböző dolog. A naplózó fájlrendszereket (például `ext3`) azért fejlesztették ki, hogy egy váratlan rendszerleállás után a fájlrendszer egységességét gyorsan helyre lehessen hozni. Az `fsck` ugyanis akár több órán keresztül is elszöszmötölhet a fájlrendszerrel, ami például egy kiszolgáló esetében elég kellemetlen lehet. A gyors helyreállítást úgy éri el, hogy minden írási műveletet először csak egy adatbázisba (a naplóba) ír bele, majd ha a lemeznek már nincs jobb dolga, a művelet csak akkor hajtódik végre valóban. Ha a rendszer esetleg összeomlana, mielőtt a naplóban feltüntetett műveletek befejeződtek volna, akkor a rendszer újbóli indulásánál elég csak a naplót megvizsgálni, hogy volt-e félbemaradt művelet. Ha volt, akkor azt fél másodperc alatt el lehet végezni, és nem kell az egész fájlrendszer összes fájlleíróját megvizsgálni. Ez ugyan nagyon hasznos dolog, csak épp semmi köze sincs a naplószerkezetű fájlrendszerhez. Linuxhoz sajnos nem ismerünk megbízhatóan használható LFS-megvalósítást. Bár számos projekt célul tűzte ki a megvalósítását, a legtöbb a fejlesztés korai szakaszában abbamaradt.
A FreeBSD viszont támogatja az LFS-t.

A védelem kérdése

Az operációs rendszernek nemcsak adataink tárolásáról, de védelméről is gondoskodnia kell. Pontosabban arról, hogy meghatározható legyen, hogy ki mihez férhet hozzá, illetve milyen műveleteket hajthat végre például egy állományon (esetleg csak olvashassa, de ne írhassa felül stb.). Ennek módját védelmi mechanizmusnak nevezzük, ami szigorúan technikai feladat.

Ez az az elv, amelynek alapján az operációs rendszer a védelmi adatokat tárolja. Egy védelmi elv akkor jó, ha egyrészt nem játszható ki. (Például nincs benne olyan kiskapu, aminek a segítségével a felhasználó képes olyan dolgot megtenni, amit neki nem lenne szabad. Ha valamiféle programhiba miatt történne mindez, az nem számít. Attól, hogy a megvalósítás rossz, az elv még jó lehet.) Másrészt szabadságot is kell biztosítania ahhoz, hogy úgy alakíthassuk ki az erőforrások védelmét, ahogy az nekünk tetszik. Nem jó az a védelmi mechanizmus, amelyik csak tiltást és engedélyezést valósít meg, mivel ennél sokkal finomabb árnyalatokra is szükség lehet. Például előfordulhat, hogy jó lenne, ha Józsi csak olvashatna az adott fájlból, míg Pisti csak írhatna bele, és Bélának semmiféle jogosultsága nem lenne hozzá.

Védelmi mechanizmusok

Először is mi az, amit egy rendszerben védenünk kell? Védenünk kell magát a gépet: ide soroljuk a processzort (például egy folyamat ne terhelhesse korlátlanul), a memóriaszakaszokat (az egyik folyamat ne írhasson a másik memóriaterületére) vagy a nyomtatót (ne minden felhasználó használhassa, illetve a nyomtatási sorból ne törölhesse ki más nyomtatnivalóját). Óvnunk kell továbbá a programot is – ez alatt a folyamatokat, az állományokat és az adatbázisokat értjük. A védelmezendő eszközök összességét és a programokat együttesen objektumoknak nevezzük.
Az objektumok fontos tulajdonsága, hogy nevük van. A név a rájuk való hivatkozást szolgálja, ezért egyedi, azaz nem lehet még egy ilyen nevű objektum a rendszerben. Minden objektumhoz tartozik még egy úgynevezett műveleti lista, ami azokat a műveleteket tartalmazza, amelyeket az adott objektumon végre lehet hajtani. Egy állomány esetében ilyen művelet lehet az olvasás, írás, végrehajtás, átnevezés stb.
Azzal, hogy erőforrásainkat objektumokba csoportosítjuk, pontosan meghatározhatjuk, hogy mely folyamatok mit csinálhatnak, illetve mit nem csinálhatnak az adott objektummal. Ezek a folyamat adott objektumra érvényes jogosultságai. A védelmi mechanizmus tehát nem más, mint az a módszer, aminek a segítségével megtiltható, hogy egy folyamat elérhesse azokat az objektumokat, amelyekhez nincs jogosultsága.
A dolog azonban nem merül ki ennyiben, ugyanis léteznek olyan objektumok, amelyekhez a folyamat ugyan hozzáférhet, de nem hajthatja rajta végre az összes műveletet. Például olvashat egy állományt, de írni nem tud bele.
Ebben idáig még nincs semmi misztikus, teljesen hétköznapi dolog, hogy egyes állományokat olvashatunk, de nem írhatunk, vagy fordítva. Most azonban egy kicsit „vadabb" fogalmat vezetünk be azért, hogy könnyebben megérthessük, milyen a jó védelmi mechanizmus. De nem kell megijedni, ez sem lesz teljesen ismeretlen, inkább csak szokatlanabb megvilágításba helyezi a dolgokat. Ez a fogalom pedig a védelmi tartomány, ami alatt egy objektumot és a rá vonatkozó jogosultságokat értjük.
A jogosultságok alatt most azokat a műveleteket fogjuk össze, amelyeket egy folyamatnak végre szabad (azaz jogában áll) hajtania.
A folyamatok jól meghatározott védelmi tartományokban futnak. Hogy melyik objektumhoz férhetnek hozzá, illetve melyik objektumon milyen műveleteket hajthatnak végre, az attól függ, hogy éppen melyik védelmi tartományban tartózkodnak. A legtöbb rendszerben a folyamatok futás közben átugorhatnak az egyik védelmi tartományból a másikba. Ennek rendje és módja minden típusú rendszerben más és más.
Minden folyamat rendelkezik egy olyan jellemzővel, amelyik egyértelműen meghatározza a pillanatnyi védelmi tartományt. A Unix esetében ez az `uid` (User ID, felhasználó azonosító) és a `gid` (Group ID, csoportazonosító). Ezekkel a korábbi részek során már találkozhattunk. A Unix minden felhasználóhoz, illetve csoporthoz egy egyedi számot rendel, ami alapján azonosítja azt. A felhasználói név csak a felhasználók számára fontos (mivel egy nevet könnyebben meg lehet jegyezni, mint egy számot), a rendszer számára azonban minden felhasználó csak egy szám.
A folyamat `uid`-je és `gid`-je azt mondja meg, hogy melyik felhasználó, illetve csoport jogosultságaival bír.
Ezért az objektumokra vonatkozó jogosultságokat nem a folyamatokra, hanem a felhasználókra és a csoportokra határozzuk meg. Így az `uid` és a `gid` egyértelműen megadja az adott folyamat védelmi tartományát. Magyarán, ha két folyamat `uid`-je és `gid`-je megegyezik, akkor pontosan ugyanolyan jogosultságok vonatkoznak rájuk.
Nem biztos, hogy egy folyamat a futása során végig ugyanabban a védelmi tartományban marad, bizonyos esetekben valamiképpen át kell lépnie egy

TARTOMÁNYOK	File1	File2	File3	File4	Nyomtató1
1.	Olvasás, Írás		Olvasás		
2.		Olvasás, Írás, Végrehajtás		Olvasás	Írás
3.	Olvasás, Végrehajtás				Írás

1.

TARTOMÁNYOK	File1	File2	File3	File4	Nyomtató1	T.1	T.2	T.3
1.	Olvasás, Írás		Olvasás					
2.		Olvasás, Írás, Végrehajtás		Olvasás	Írás	Belépes		
3.	Olvasás, Végrehajtás				Írás			

2.

másikba. Erre a legkézenfekvőbb példát a `SETUID`-es, illetve a `SETGID`-es programok nyújtják. A Unix lehetőséget ad arra, hogy az `EXEC` rendszerhívás segítségével elindított program a tulajdonosának, és nem az őt futtatónak a jogosultságával fusson. Ilyen például a `passwd` nevű program, amelynek segítségével a felhasználók megváltoztathatják a jelszavukat. Ehhez az kell, hogy a ***/etc/passwd*** (illetve a mai rendszerekben a ***/etc/shadow***) állományt írhassák. Ezt azonban – érthető okokból – kizárólag a rendszergazda teheti meg, tehát a `passwd`-nek mindenképp `SETUID`-esnek kell lennie. (Monolitikus rendszerekben akkor is megváltozik a védelmi tartomány, amikor a folyamat egy rendszerhívást hajt végre. Ebben az esetben a folyamat felhasználói módról magmódra vált. A magmódban alig-alig van olyan objektum, amelyhez ne férhetnénk hozzá korlátlanul, így nyilván a védelmi tartomány sem lehet ugyanaz, mint a felhasználói módban.)

Hogy mindez valóra válhasson, az operációs rendszernek gondoskodnia kell a védelmi tartományok tárolásáról, különben nem tudná eldönteni, hogy a folyamat végrehajthatja-e az adott műveletet. Hogy ez miként történjen meg, azt határozza meg a védelmi mechanizmus.

Védelmi tartományok nyilvántartása

A legkézenfekvőbb megoldásnak az tűnhet, ha az egészet egy táblázatban tároljuk (*1. ábra*). Minden oszlop egy-egy objektumnak, a sorok pedig a tartományoknak felelnének meg. Ezáltal a táblázat minden eleme az adott objektumra vonatkozó megengedett (vagy más néven legális) műveleteket fogja tartalmazni, amelyeket az adott folyamat végrehajthat az objektumon. Az ilyen táblázatot védelmi mátrixnak nevezzük. Vessünk egy pillantást a *2. ábrára* is! Itt a dolgon csavartunk azáltal, hogy magukat a tartományokat is objektumoknak tekintjük, csak a rajtuk értelmezhető művelet nem éppen az írás, az olvasás, illetve a végrehajtás, hanem a belépés. Ez a művelet azt jelenti, hogy az összes első tartományban lévő folyamat átlépheti ebbe (a jelen esetben a második) védelmi tartományba. Láthatjuk tehát, hogy ez a táblázatos módszer remekül alkalmazható azokban az esetekben is, amikor például `SETUID`-es programokat futtatunk.

A védelmi mátrixok egyszerűen használhatóak és könnyedén átültethetőek lennének a gyakorlatba is, a tartományok nyilvántartására azonban sehol sem használják. Ennek az az oka, hogy rendkívül sok objektum létezik, és közülük a legtöbbhöz a folyamatoknak szinte semmiféle jogosultságuk nincs. Ebből adódóan a védelmi mátrix hihetetlenül nagy és „szellős” lenne. Nem gazdaságos megoldás teljes egészében tárolni egy olyan táblázatot, amelyik nagy és majdnem üres. Sokkal célravezetőbb, ha a védelmi mátrixot soronként vagy oszloponként tároljuk, olyan módon, hogy csupán a nem üres elemeket jegyezzük meg.

Hozzáférést vezérlő lista (ACL)

Kezdjük az oszloponkénti tárolással. Mivel mi csak a nem üres elemeket kívánjuk tárolni, az egészet úgy is felfoghatjuk, mintha minden objektumhoz egy tartományokból álló listát rendelnénk. Ebben a listában azonban csak azokat a tartományokat tüntetjük fel, amelyeknek van valamiféle jogosultságuk az adott objektumra. Ha az 1. ábrán látható példát vesszük alapul, akkor a `File1` nevű objektumhoz tartozó listában az első és a harmadik tartomány szerepelne. Ha ehhez a listához hozzávesszük a tartományokhoz tartozó elérési módot is (tehát azoknak a műveleteknek a halmazát, amelyet a kérdéses objektummal elvégezhetünk), akkor azt ACL-nek (Access Control List, azaz hozzáférést vezérlő listának) nevezzük.

Erre láthatunk egy példát a *3. ábrán* (megint csak az 1. ábrából kiindulva). Mivel (Unix esetén) a védelmi tartományokat az `uid` és a `gid`, azaz a felhasználó és annak csoportja határozza meg, a lista minden eleme három részből tevődik össze: a felhasználó, a csoport és az objektumon végrehajtható műveletek.

Az ACL ennél sokkal többet rejt magában. A *4. ábrán* újabb példát láthatunk ACL-ekre. Semmi akadálya sincs annak, hogy egy hozzáférést függetlenítsünk a `uid`-től vagy `gid`-től. Példánkban ezt a * (csillag helyettesítőkarakter) jelöli. Ahol ez található az `uid`, illetve a `gid` helyett, ott nem érdekes, hogy az adott folyamat melyik `uid`-be, illetve `gid`-be tartozik.

Az ACL igazi előnye azonban az utolsó sor esetében mutatkozik meg. Könnyedén megtehetjük, hogy valakit anélkül zárjunk ki egy bizonyos jogosultságból, hogy a vele egy csoportban lévő felhasználókat a dolog érintené. Az itt leírtakkal egy gond akad, hogy a Unix nem használ ACL-t, illetve mégis, de másmilyet, mégpedig egy kilenc bites „változatot”. Ez alatt azt értjük, hogy minden objektumhoz (a gyakorlat nyelvére átültetve: minden állományhoz) három darab „rwx bitet” rendel: egyet a fájl tulajdonosához, egyet a csoportjához és egyet mindenki máshoz. Kár tagadni: ez nagyon messze áll egy teljes értékű ACL-es megoldástól. Ez a Unix egyik komoly hátránya. Noha a különlegesebb eseteket kivéve ennyivel is elboldogulunk, de

TARTOMÁNYOK	File1	File2	File3	File4	Nyomtató1
1. Józsi, tanuló	Olvasás, Írás		Olvasás		
2. Pityu, tanár		Olvasás, Írás, Végrehajtás		Olvasás	Írás
3. Samu, tanuló	Olvasás, Végrehajtás				Írás

3.

TARTOMÁNYOK	File5	File6	File7
1. Józsi, tanuló	Olvasás, Írás		Olvasás
2. Pityu, tanár		Olvasás, Írás	
3. Samu, tanuló			

4.

TARTOMÁNYOK	File1	File2	File3	File4	Nyomtató1
1.		Olvasás, Írás, Végrehajtás			
2.				Olvasás	
3.					Írás

5.

ha sok (több száz vagy ezer) felhasználónk van, akkor a felügyelet rendkívül költséges feladat. Ilyenkor érdemes elgondolkodni, nem járnánk-e jobban egy olyan operációs rendszerrel (például VMS), ahol egy kicsit „finomabban” is beállíthatjuk az egyes objektumokra vonatkozó jogosultságokat.
Az ACL-nek akad azonban hátránya is: ha egy objektum ACL-ét megváltoztatjuk, akkor az új „jogszabály” még nem fog azokra a folyamatokra vonatkozni, amelyek már megnyitották az adott állományt.

Képességi listák

A másik módszer a védelmi mátrix soronként történő tárolása. Ez tulajdonképpen az ACL fordítottja: nem azt tároljuk, hogy egy objektummal ki mit csinálhat, hanem azt, hogy ki melyik objektummal mit csinálhat. Némileg szakszerűbben megfogalmazva: a futó folyamatokhoz rendeljük hozzá az általa elérhető objektumokat a megengedett műveleteivel együtt (azaz amit jogosultsága van végrehajtani). Ez a képességi lista, amelynek elemei a képességek.
Tekintetünket szegezzük az *5. ábrára*, amelyen egy képességi listára láthatunk példát. Fontos, hogy a képességi lista az összes olyan műveletet magában foglalja, amit a folyamat végrehajthat. Annak sincs semmi akadálya, hogy a képességi listákra is objektumként tekinthessünk, amelyek más képességi listába (ez az úgynevezett tartománymegosztás) felvehetők.
Hogy egy folyamat egy rendszerben mit tehet meg és mit nem, az kizárólag a képesség listájának tartalmától függ. Könnyen belátható, milyen súlyos következménnyel járna, ha a folyamat némi ravaszkodás árán átírhatná a saját (vagy éppen egy másik folyamat) képességi listáját. Ilyesmit nem szabad megengedni, ezért a képességlistákat úgy kell tárolnunk, hogy a folyamatok még véletlenül se babrálhassanak vele. Erre több mód is kínálkozik: a legkézenfekvőbb az, hogy az operációs rendszer memóriaterületére tesszük, és a folyamatok a sorszámukkal hivatkoznak a képességekre. Kényelmesebb azonban, ha hardveresen oldjuk meg a feladatot, igaz, ehhez egyedi szolgáltatásra van szükség, ami elég kevés kiépítésben létezik. (Az eszközből történő megvalósítás úgy működne, hogy minden memóriaszóhoz egy bit lenne rendelve, ami azt mondja meg, hogy az adott szó tartalmaz-e képességet vagy sem.)
A leghatékonyabb megoldás azonban az lenne, hogy a képességlistát a folyamat saját területén tárolnánk. A védelmét úgy szavatolnánk, hogy a rendszer titkosítaná egy olyan kulccsal, amit kizárólag csak ő ismer. Kellően erős titkosítás használatakor nem fenyegetne minket az a veszély, hogy esetleg valamelyik folyamat feltöri, viszont az is igaz, hogy sokkal erőforrás-igényesebb megoldás, mint az előző kettő.
A biztonság mellett fontos kérdés még a jogosultságok beállítása, azaz miként lehet megadni, hogy ki mihez férhessen hozzá. Erre az a megoldás, hogy minden objektum rendelkezik úgynevezett általános jogosultságokkal, amelyek szintén kiadhatók a többi objektumfüggő jogosultságok (olvasás, írás, végrehajtás stb.) mellé. Az általános jogosultságok egyike lenne például egy új képesség létrehozása vagy éppen törlése.
A képességi listás megoldás elsősorban osztott rendszerek esetében lehet kényelmes, de ennek is megvan a maga ára. Például sokkal nehezebb feladat egy jogosultság visszavonása. Ennek az az oka, hogy itt az objektumokhoz tartozó képességek a lemezen összevissza találhatók, míg az ACL esetében csupán egyetlenegy módosításra volt szükség.
Most már tudjuk, hogyan tárolja a rendszer azt, hogy melyik felhasználó mihez férhet hozzá. Ám a biztonság kérdése ennél sokkal messzebbre nyúlik. A következő részben éppen ezért a felhasználók azonosításával kapcsolatos kérdésekről, a jelszavakról, programhibákról, vírusokról és férgekről, rejtett csatornákról lesz szó, és még sok minden másról.

Garzó András (garzoand@interware.hu) Körülbelül három éve foglalkozik Linux- és más Unix-rendszerekkel. Legjobban az operációs rendszerek lelkivilága érdekli, de nyitott egyéniség. Kedvenc étele a palacsinta, és van egy Richard nevű macskája. Minden észrevételt, megjegyzést, levelet szívesen fogad.