Beköszöntő

Nyár van, uborkaszezon, a fű se nő. Nekünk valahogy mégsem okozott problémát a címlapsztori kiválasztása. Jelentjük: 2005. július 6-án az Európai Parlament 648 nem szavazattal, 14 ellenében, 18 tartózkodás mellett elvetette a számítógéppel megvalósított találmányok szabadalmazására vonatkozó (ismertebb nevén: szoftverszabadalmi) irányelvtervezetet.
Nyugodjék békében. Aki pedig kíváncsi szegény pórul járt törvényjavaslat tündöklésének és bukásának történetére, nos az elolvashatja ***Dr. Dudás Ágnes*** „nekrológját".
Augusztusi számunkban egyelőre még dominálnak a komolyabb szakmai tartalommal bíró írások, de – nem kis örömünkre – az előző számunkban közzétett felhívásra, melyben a lap arculatváltásához új cikkírókat kerestünk sokan jelentkeztek. A weblapunkon közzétett témalista (➲ ***http://www.linuxvilag.hu/szerzoknek***) így azóta számos új elemmel bővült, illetve szaporodnak a téma „foglaltságát" jelző pluszok is.
Jó szerzőből természetesen soha nem elég, így továbbra is várjuk a javaslatokat, észrevételeket, de legfőképpen a lelkes cikkszerzők jelentkezését a *szerkesztoseg@linuxvilag.hu* címen.
Régi szerzőink sem pihentek. Több korábban indult sorozat is folytatódik ebben a számban. ***Auth Gábor*** a ***FreeBSD***, ***Fülöp Balázs*** a ***Java*** programozás, ***Komáromi Zoltán*** a ***PEAR***, ***Garzó András*** pedig a számítógép-hálózatok alapjainak bemutatásával lépett tovább, illetve befejező részéhez érkezett ***Fábián Zoltán eFltk***-ról szóló sorozata.
Többen kérdezték, így itt is megerősítjük: ezek a sorozatok természetesen a megújult lapban is folytatódnak. Ugyanakkor ahogy ígértük, szándékunk szerint szeptembertől egyre gyakoribbak lesznek a kifejezetten felhasználóknak szóló tartalmak is.

Addig is kellemes időtöltést,
jó szórakozást kíván
a Linuxvilág stábja.

Belekotyogók fóruma

A *SourceLabs* új – vagy inkább újabb –, közösségi, nyílt forrású fejlesztési tervezeteket összefoglaló webhelyet indított *Swik* névvel. A *SourceLabs* nyílt forrású programokra alapuló tanácsadói és támogatási tevékenységet végez, nyilatkozatuk szerint több ügyfelüktől is kaptak olyan visszajelzést, miszerint nincs olyan webhely, ahol teljes értékű megoldások összeállításához kényelmesen lehetne nyílt alkalmazásokat gyűjteni. A többi hasonló webhely közül a *Swik* azzal próbál kitűnni, hogy *Wiki* jellegű eszközöket biztosít a fejlesztőknek, akik a rendszeren belül tárolt tervezetek bármelyikének adataiba belenyúlhatnak, módosíthatják és frissíthetik azokat. A hozzászólások azonnal meg fognak jelenni a rendszerben, és a cég nem fogja moderálni azokat – ez első hallásra veszélyes lehetőségnek tűnik, de a *SourceLabs* tapasztalatai szerint, néhány deviáns személyt leszámítva, a közösség képes önmaga szabályozására. A webhely további szolgáltatása lesz az *RSS* alapú értesítés az egyes tervezeteket érintő változásokról. A *Swik* egyelőre kevesebb, mint ezer tervezetet tartalmaz, ám a várakozások szerint a közeljövőben ez a szám meredeken nőni fog.

➲ www.swik.net

Csere-bere az Operában

Az *Opera Software* bemutatta *BitTorrent* ügyféllel ellátott böngészőjének előzetes változatát. A 8.02-es *Operát Windows*, *Linux* és *Mac OS X* alá tölthetjük le, és ez az első olyan böngésző, amely önmagában, külön ügyfélprogram nélkül teszi lehetővé az egyenrangú fájlcsere-hálózat használatát. Az *Opera* fejlesztői ezzel veszélyes vizekre eveztek: a program ilyen irányú bővítése a felhasználók körében ugyan népszerű lehet, ám a jogvédelemért felelős szervezetek és a „komoly” cégek körében aligha; a rajta folyó illegális zene-, film- és programcsere miatt a kiadók és a hatóságok az utóbbi időben komolyan felléptek a *BitTorrent* hálózat ellen.

➲ www.opera.com

iPod Linux

Cinikusan mondhatnánk, az *iPodot* is utolérte a sorsa: készült hozzá *Linux*-terjesztés. Az *iPod Linux* állítólag – a legújabb modellek kivételével – könnyedén telepíthető, mindössze 5 MB tárhelyet igényel, és természetesen elérhetően hagyja a készülék eredeti operációs rendszerét is. Legfontosabb előnye, hogy a gyári szolgáltatásokkal ellentétben kiváló minőségű, akár 16 bites, 96 kHz-es, sztereó hangfelvételek készítését is lehetővé teszi. Az *iPod Linux* képes a *JPEG*, a *GIF* és

a *BMP* fájlok fekete-fehér megjelenítésére, több egyszerű játékot, valamint számológépet és naptárat is tartalmaz. Fejlesztése jelenleg is folyik, hamarosan *iPod-iPod* hálózatok létrehozására, *Game Boy* emulátor futtatására és *Doomozásra* is alkalmas lesz. Fontos megjegyezni, hogy az *iPod Linux* hanglejátszási szolgáltatásának minősége az *uClinux* korlátai miatt egyelőre elmarad az eredetitől.

➲ www.ipodlinux.org

Mint az úthenger

Alig fejezte be a brazil *Conectiva* felvásárlását, illetve a névváltást, az ex-*Mandrakesoft Mandriva* máris újabb

linuxos vállalatot olvasztott magába: a *Lycorist*. Az amerikai székhelyű *Lycoris* különösen felhasználóbarát, elsősorban asztali gépekre készített terjesztéseiről ismert, felvásárlása valószínűsíthetően – a *Conectiva* megszerzéséhez hasonlóan – a piacszerzést szolgálja. A *Lycoris* terjesztései *Észak-Amerikában* nagy népszerűségnek örvendenek, és széles körben kaphatók mind dobozos formában, mind számítógépekhez mellékelve. A *Mandrakesoft*, mint emlékezetes, tavaly tavasszal még csődvédelembe volt kénytelen menekülni, most viszont minden jel szerint az előre menekülés és az agresszív terjeszkedés stratégiáját választották – igaz, ebbe is belebukott már néhány vállalat. A *Lycoris* beolvasztása alighanem könnyen ment, ugyanis az amerikai vállalat is komoly gondokkal küzdött, az utóbbi időkben több fejlesztését is meg kellett szakítania, illetve munkatársainak számát is csökkentette. A felvásárlás nyomán az eljövő években várhatóan különféle részeket fognak átemelni a terjesztések között, majd a cégek operációs rendszerei és szolgáltatásai fokozatosan összeolvadnak.

➲ www.mandriva.com

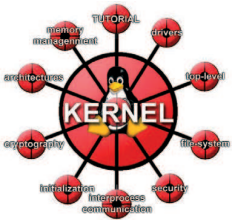
➲ www.lycoris.com

Kattintani tud? Plusz egy pont

A *California State University* segédrektora, *Lorie Roth* fontos felfedezést tett: attól, hogy a diákok mindennapjaik során számtalan feladatra használják az internetet, még nem feltétlenül képesek az általa kínált lehetőségek valóban hatékony kiaknázására. Bepötyögni a bennünket érdeklő téma kulcsszavait a *Google*-be, elektronikus leveleket írni ismerőseinknek, *MP3* fájlokat töltögetni – ezek kétségtelenül közelebb visznek ahhoz, hogy megbarátkozzunk a számítógéppel, ugyanakkor nem hozzák magukkal a megbízható információk felkutatásának, illetve a talált adatok ellenőrzésének képességét. Márpedig ezek a képességek manapság ugyanolyan fontosak, mint a számolás, az olvasás vagy éppen az írás képessége – véli az egyetemi vezető. Éppen ezért kidolgoztak és már „tesztelnek” is egy tesztet, amellyel a diákok informatikai-technológiai-olvasási képességeit, összefoglaló néven *Internet IQ*-ját kívánják felmérni. Az eredményeket hamarosan arra is fel kívánják használni, hogy az egyetemre felvett, e tekintetben társaikhoz képest lemaradóban lévő diákokat rövid továbbképzésre küldhessék. A dolog fontosságát mi sem jelzi jobban, mint az az adat, mely szerint az amerikai diákok már évek óta többet használják az internetet, mint a hagyományos könyvtárakat.

Magmodell

A *Software Revolution* elkészült a 2.6-os *Linux* rendszermag részletes, *UML 2.0* alapú modelljével. Az összekapcsolt,



szabadon méretezhető, vektorgrafikusan ábrázolt elemekből – folyamatábrákból, szerkezeti diagramokból, állapotgépekből, hívási táblázatokból, döntési táblákból stb. – álló modell és a hozzá tartozó szöveges leírások a rendszermag minden fontosabb alrendszerét lefedik, ide értve a forráskódban található mezőket és függvényeket is. A cég szerint ez a *Linux* rendszermagról valaha készült legteljesebb és legrészletesebb modell; melyet egyébként egy *SVG*-nézegető beépülő modul birtokában bárki szabadon böngészhet. A programozás iránt érdeklődőknek érdemes lehet akár csak kíváncsiságból belenézniük, nagy kapufát nem lőhetnek vele.
➲ http://www.softwarerevolution.com/jeneral/open-source-docs.html

GeForce 7800

A grafikus vezérlők világában ma még különlegességnek számító újdonságokat hoz magával az *NVIDIA* hamaro-

san az üzletekben is megjelenő *GeForce 7800 GTX* vezérlője. Bár szolgáltatásaiban kevéssé fog különbözni a 6800-as sorozat tagjaitól, felépítésében már jóval fejlettebbnek mutatkozik náluk. A lapka egyik érdekessége, hogy az egyes részei képesek lesznek egymástól eltérő órajelekkel üzemelni, így például a képpont- és a raszterfeldolgozó 430, míg a vertexkezelő részegység 470 MHz-en üzemel. A műszaki csemegék iránt kevésbé fogékonyakat persze nyilván inkább az érdekli, hogy az alapórajel módosításával a másik órajel is módosul, vagyis e tekintetben nem kell a tuningolási lehetőségek elvesztésétől tartani. A 24 darab képpontfutószalag, a 8 vertexfutószalag, a 300 millió tranzisztor és a jelenlegi csúcsmodell *GeForce 6800 Ultrához* képest mért 50 százalékos teljesítménybővülés pedig magáért beszél.
➲ www.nvidia.com

Qt 4

Újabb mérföldkőhöz érkezett a norvég *Trolltech*: elkészült ismert keresztplatformos fejlesztőeszközének, a *Qt*-nek legújabb, 4-es változatával. A *Qt 4* segítségével összetett, ügyfél- és kiszolgálóoldali alkalmazásokat egyaránt lehet fejleszteni, a *Microsoft Visual Studio .NET*-tel egybeépítve, illetve az új licencelési módozat révén pedig a kereskedelmi, windowsos alkalmazásokat készítők is könnyebben kihasználhatják lehetőségeit. A *Qt 4* új, nagyteljesítményű, szükség esetén lecserélhető grafikus motort kapott; kiterjesztették többszálú futtatási képességeit; valamint javítottak a sebességén és a memóriakezelésén. A *Qt 4* a *GPL* fejlesztések számára ingyenesen, a kereskedelmi munkákhoz pedig fizetős változatban máris elérhető, összesen háromféle – *Qt Console*, *Qt Desktop Light* és *Qt Desktop* – variációban.
➲ www.trolltech.com

Beért az Nvu

Végre-valahára elkészült a nyílt forrású weblapszerkesztő, az *Nvu 1.0*-s változata. A hónapok óta béta változatok formájában készülődő kiadás olyan ismert alkalmazások babérjaira tör, mint a *Microsoft FrontPage* és a *Macromedia Dreamweaver*. A *WYSIWYG*, vagyis „amit látsz, azt kapod” jellegű szerkesztő készítői mindent meg is tettek, hogy méltó vetélytársai legyenek a nagyoknak, az *Nvu 1.0* kiváló stabilitással és teljesítménnyel, beépített helyesírás-ellenőrzővel, *CSS*-szerkesztővel és kiterjedt súgóval rendelkezik, szigorúan igazodik a *HTML 4.01* és az *XHTML 1.0* szabvány előírásaihoz, valamint a lehető legtisztább kódot állítva elő böngészők széles választékával biztosítja az együttműködést. Az *Nvu 1.0* több nyelven is elérhető, illetve nemcsak *Linux*, de *Windows*, *FreeBSD*, *OS/2* és *Mac OS X* alatt is futtatható.
➲ www.nvu.com

Medgyesi Zoltán
(mz@rettesoft.hu)
A Linuxvilág hírszerkesztője. Szabadidejét legszívesebben a barátnőjével tölti, szeret autózni és bográcsban főzni.

Mi újság a rendszermag fejlesztése körül?

A *Linux* rendszermag fejlesztésében jelenleg egyre inkább előtérbe kerül az üzembiztosság kérdése. *Greg Kroah-Hartman* és *Chris Wright* önkéntesen elvállalta egy szigorúan ellenőrzött üzembiztos fa gondozását. Régebben az üzembiztos és a fejlesztői fák éves hosszúságú ciklusokban váltották egymást, az új üzembiztos fa azonban a fejlesztői mellett fog futni, és kizárólag a fontos hibajavításokat fogja tartalmazni. A *Linus Torvalds*-féle 2.6.11-es kiadást tehát egy 2.6.11.1-es és egy 2.6.11.2-es követte, illetve *Greg* és *Chris* részéről további üzembiztosságot növelő kiadások is megjelentek. A 2.6.11.z fa üzembiztosságának növelése a 2.6.12 kiadása után is folytatódni fog, ahogy majd lesz egy 2.6.12.1-es kezdetű sorozat is. A korábbi üzembiztos sorozatokkal ellentétben, amelyeknek karbantartói meglehetősen nagy szabadsággal rendelkeztek a tekintetben, hogy mely foltokat kívánják befogadni, most szigorú szabályok határozzák majd meg, hogy mi kerülhet be az üzembiztos ágba; sőt, azt is pontosan meg fogják határozni, hogy mennyi időnek kell eltelnie a foltok beadása és elutasítása vagy elfogadása között. *Linus* az üzembiztos fát egyszerűen „szivatós" fának nevezte, hiszen nincs olyan ép elméjű ember, aki szívesen felvállalta volna a karbantartásával járó gondokat. *Chris* és *Greg* azonban nem rettentek el a kihívástól, bár a pontos folyamatok egyelőre csak kialakulóban vannak. Úgy tűnik azonban, hogy az üzembiztosság ismét kiemelt szerepet kapott a *Linux* fejlesztésében.

A *SysFS* fájlrendszer sorsa az utóbbi időben kissé bizonytalanná vált. Fejlesztését elsősorban az ösztönözte, hogy meg kellett szabadulni a *ProcFS* régről maradt terhétől, illetve a környékén uralkodó fejetlenségtől, és valami letisztult, ésszerű dolgot kellett – volna – a helyére illeszteni. A fejlesztők remélték, hogy tiszta lappal indulva a régi hibákat el lehet majd kerülni. Nemrég azonban a rendszermaggal foglakozók rájöttek, hogy a *SysFS* könyvtárainak egyike rossz helyre került: a */sys/block* könyvtárnak valójában */sys/class/blocknak* kellene lennie! Hiába, késő! Időközben komoly mennyiségű felhasználói kód készült úgy, hogy a könyvtár meglévő elérési útját vette figyelembe. *Greg Kroah-Hartman*, ha vonakodva is, de kénytelen volt beismerni, hogy a *SysFS* inkonzisztenciáját nem lehet kijavítani. A *SysFS* szeplőtlen arcán megjelentek az első ráncok.

A *SquashFS* továbbra is igyekszik a rendszermagba, illetve próbálja teljesíteni az elvárásokat, miközben vezető fejlesztője, *Phillip Lougher* egyre inkább gyűlöli a világot. Az előrelépés egyik akadálya talán inkább a kultúrák eltérőségéből fakad: a rendszermag fejlesztői valamilyen meggyőző érvet szeretnének hallani a beépítés mellett, míg *Phillip* inkább a kódolás, semmint az „eladás" iránt érdeklődik. A többi akadály inkább műszaki jellegű. Például jelenleg a *SquashFS* legfeljebb 4 GB-os fájlok kezelésére képes. Persze, mivel a *SquashFS* tömörített fájlrendszer, a valóságban ez inkább 8 GB-nyi adatot jelent. Nemrég az is kiderült, hogy a *readdir()* sem a *./*, sem a *../* könyvtárat nem adja vissza a *SquashFS* esetében, ahogy azt gyakorlatilag az összes szabványos fájlrendszernél megszokhattuk. Ezek a problémák – továbbiak mellett – sajnos komolyan hátráltatják a *SquashFS*-t a normál rendszermag részévé válásban.

A *FUSE (Filesystem in USErspace, fájlrendszer felhasználói térben)* várhatóan bekerül a fő rendszermagfába, miután időzött egy kicsit *Andrew Morton -mm* ágában. A *FUSE* sok mindenen ment már keresztül, *Linus Torvalds* például sokáig úgy tartotta, hogy egy felhasználói térben futó fájlrendszer eleve elvetélt ötlet, soha semmi értelmes nem fog kisülni belőle. A *FUSE* ennek ellenére tetszetős motorrá fejlődött, és a jelek szerint ellenzői is elcsitultak. *Andrew* is pártolja, és számíthatunk rá, hogy *Linus* előtt is támogatni fogja.

Több tervezet is gazdát cserélt az utóbbi időben, illetve egyes programok karbantartói mostanában nyertek hivatalos elismerést. *Pete Zaitcev* lett az *USB* blokkos illesztőprogram és a *Yamaha PCI* hangillesztőprogram hivatalos gondozója. *Herbert Xu* átvette *James Morris* társgondozói helyét a rendszermag *crypto API*-jának fejlesztésében. *Gerd Knorr* felhagyott a *Video4Linux* karbantartásával, amivel a tervezet pillanatnyilag gazdátlanul maradt.

Zack Brown

Linux Journal 2005. július, 135. szám

Játékok a vásárban

A 4. Információtechnológiai és telekommunikációs vásár idén megújult formában kerül megrendezésre a Budapesti Nemzetközi Vásárral egy időben. Tavaly több mint 120 ezres közönség érkezett a BNV-re a HUNGEXPO Budapesti Vásárközpontba.

Nagy érdeklődés mellett zajlottak a tavalyi *INFOmarket* PC-s és konzolos bajnokságai is, ezért idén – 2005. szeptember 17. és 25. között – a látogatókat minden eddiginél több játék és bajnokság várja kilenc napon át.

Vadászpilóták bajnoksága

Az *INFOmarketen* központban szerepel a *GAME* szekció, a játékok nagy választékával. A közönség 2005-ben először kísérheti figyelemmel a repülőgép szimulátorok bajnokságát a „B" pavilonban. A *Jet-Fly Légiharc Kupa* budapesti döntőjére szeptember 23-25. között kerül sor. A szeptember 17-19-i előselejtezők, valamint a bajnokság izgalmas pillanatait a látogatók kivetítőről nézhetik. A jelentkező vadászpilóták további információkat találnak a ➲ www.jetfly.hu honlapon.

FIFA 2005 – World Cyber Games

A hétköznapokon mindenki kipróbálhatja a játékokat, a hétvégéken, pedig szurkolhat a bajnokságok résztvevőinek: az első hétvégén a *World Cyber Games – Counter-strike offline* selejtezők versenyzőinek, a következő hétvégén, pedig a látogatók részesei lehetnek a *FIFA 2005* Bajnokság szoros mérkőzéseinek. Hasznos tudnivalók a jelentkezésről megtalálhatóak: a ➲ http://hu.worldcybergames.com, és a ➲ www.fifahungary.com honlapokon.

Újdonságok a kínálatban

A kiállításra elhozza újdonságait többek között Magyarország legnagyobb konzolforgalmazója. A *GAME* szekció mellett a *KLASSZIKUS*, azaz a hagyományos kiállításon a felhasználók megismerhetik a legmodernebb vírusirtó programokat, irodai-, ügyviteli- és nyelvoktató szoftvereket. Ezen kívül megtalálhatóak a hagyományos hardvertermékek, a legújabb *MP3* lejátszók, valamint vetítéstechnikai berendezések.

„Gépészek" a fedélzeten

Az *INFOmarket-INFOtrenden* először kerül megrendezésre a *Magyar Tuning és Modding Kiállítás és Verseny* (a tuningbarátoknak ugyanakkor az idei már a negyedik találkozója lesz). A számítógép tervezők paradicsoma egyedi építésű számítógépekkel várja a nagyközönséget.

Kilenc nap kikapcsolódás

Az *INFOmarket* szeptember 17-25. között az egész család számára betekintést nyújt a játékok világába, az informatika és telekommunikációs piac kínálatába. A játékos bajnokságokat a párhuzamosan zajló *BNV* szórakoztató programjai egészítik ki a *HUNGEXPO Budapesti Vásárközpontban*.

További információ:
➲ www.infomarket.hu

Új termékek

WhisperStation

A *Microway* bemutatta új, elsősorban tervezési alkalmazásokhoz és kis- és középméretű fürtökbe

szánt *WhisperStation* munkaállomását. A *WhisperStation* kettő darab *AMD Opteron* vagy *Intel Xeon EM64T* processzorral, *NVIDIA FX1300 PCI Express* grafikus vezérlővel, 2 GB memóriával, különösen csendes ventilátorokkal és tápegységgel rendelkezik, valamint 20"-os *Viewsonic LCD* monitor tartozik hozzá.
A *WhisperStation* 64 bites *Red Hat*, *SUSE* vagy *Gentoo Linuxszal* vagy *Microsoft Windowszal* kapható. A vásárlók pontos és egyedi igényeinek megfelelően további bővítésére is van lehetőség, például nagyméretű merevlemezekkel, *RAID*-del és célprogramokkal.
www.microway.com

Comet12 hordozható számítógép

A *Tadpole Computer Comet12*-je egy a *Federal Information Processing Standards (FIPS) 140-2* előírásainak megfelelő, vezeték nélküli hordozható számítógép, elsősorban kormányzati használatra. A *Tadpole* vezeték nélküli, ultravékony *Sun Ray* ügyfeleinek *Comet* termékvonalára és a *Fortress Technologies* biztonságos vezeték nélküli átjáróira épülve a *Comet12* biztonságos, titkosított vezeték nélküli kapcsolatot biztosít az érzékeny adatokat is továbbító szövetségi hálózati és távközlési rendszerrel.
A *Comet12*-ben felhasznált megoldások védik a felhasználók adatait, hozzáférés-vezérlést, illetve eszköz- és készülékhitelesítést biztosítanak, valamint az adatkapcsolati rétegbeli védelemmel biztonságot nyújtanak a szolgáltatásmegtagadási támadásokkal szemben.
A *Comet12* 12,1"-os *TFT-LCD XGA* kijelzővel rendelkezik, 26,4 x 22,1 x 2,3 cm méretű, súlya pedig 1,5 kg.
www.tadpolecomputer.com

PRIMEQUEST kiszolgálók

A *Fujitsu Computer Systems* bejelentette új, *Intel Itanium 2* processzorokra épülő *PRIMEQUEST* kiszolgáló-termékvonalát.
A *PRIMEQUEST* kiszolgálók adatközpont szintű hibatűrésre képesek, és magas szintű rendszermeretezhetőséget biztosítanak az olyan az iparágban széles körben elterjedt környezetek számára, mint a *Red Hat Enterprise Linux*, a *Novell/SUSE Linux Enterprise Server* és az *Itanium* alapú számítógépekhez készült *Windows Server 2003*. A *PRIMEQUEST* kiszolgálókat felépítésüknél fogva kiváló hibatűrés jellemzi, akár nyolc magas rendelkezésre állású, teljesen független, önálló kiszolgálóként üzemeltethető, hardveresen elválasztott partíció kezelésére is alkalmasak. A *PRIMEQUEST* kiszolgálók lapkakészlete által biztosított tükrözési szolgáltatás és a rugalmas be- és kiviteli képességek szintén a magas rendelkezésre állás elérését segítik.
A *PRIMEQUEST* kiszolgálók redundáns, *SSL* protokollt alkalma-

zó felügyeleti hálózattal, beépített gigabites kapcsoló és hub összeköttetésekkel, beépített *SCSI* merevlemezekkel és a partíciók felügyeletének megkönnyítésére szintén beépített *KVM/USB* egységgel rendelkeznek.
us.fujitsu.com/computers

Orion PMC vagy PMI

A *Curtiss-Wright Controls Embedded Computing* új, kétcsatornás videótömörítő és -kibontó kártyát mutatott be *Orion* névvel. A kártya *PMC* és *PCI* foglalatba illeszkedő kivitelben egyaránt elérhető, tehát *VME*, *CompactPCI* és asztali *PCI* rendszerekbe egyaránt beépíthető. Az *Orion* kettő darab beépített *JPEG 2000* motorral rendelkezik, amelyekkel teljes méretben képes a szabványos, 625 soros *PAL* és az 525 soros *NTSC* kompozit videóanyagok kódolására. Beviteli módban az *Orion* legfeljebb tíz „egyvégződésű" vagy négy különbözeti, analóg *PAL* vagy *NTSC* videóbemenet fogadására képes, ezek közül kettőt lehet egyidejű *JPEG 2000* tömörítésre kiválasztani. Kiviteli módban az *Orion* egy vagy két *JPEG 2000* adatfolyamot fogad a 64 bites, 66 MHz órajelű *PCI* buszon keresztül, kibontja az adatfolyamokat, majd kiadja a kapott egy vagy két, független *PAL* vagy *NTSC* videojelet – erre a hátlapon elhelyezett, 20-as *MDR* aljzat szolgál. A videórögzítő és -felvevő rendszer részeként a kapott tömörített videófolyamokat helyben is el lehet menteni, de távoli megjelenítés céljából hálózaton keresztül is lehet terjeszteni. Az Orionhoz a *PowerPC* alapú *Linuxokhoz* jár illesztőprogram, illetve más rendszerekhez is léteznek kiegészítők.
Az alacsony szintű illesztőprogram- és az átfogó kártyatámogató könyvtár számos C függvényt foglal magába, ezeket többféle operációs rendszerre és géptípusra is át lehet ültetni. Az *Orion PMC* és *PCI* kivitelben érhető el.
http://www.cwcembedded.com

Linux Journal 2005. 135. szám

RTOS eszközmeghajtók illesztése beágyazott Linuxhoz

Alakítsuk hagyományosan durva és kusza RTOS kódunkat szépen formázott Linux eszközmeghajtókká.

A *Linux* viharként tört be a beágyazott rendszerek piacára. Az ipari elemzők szerint az új beágyazott 32- és 64-bites tervek fele-egyharmada a *Linuxra* épít. Több alkalmazási területet már most is a beágyazott *Linux* ural (például *SOHO* hálózatok és a képalkotó/többfunkciós perifériák), de öles léptekkel halad a tárolórendszerek (*NAS*/*SAN*), a digitális otthoni szórakoztatás (*HDTV*/*PVR*/*DVR*/*STB*) és a kézi/vezeték nélküli készülékek elsősorban a digitális mobiltelefonok piaca felé is.

A beágyazott *Linux* alkalmazások persze nem Minervát utánozva pattannak ki a fejlesztők fejéből. A legtöbb projekthez több ezer vagy akár több millió soros örökölt forráskód szükséges. Igaz ugyan, hogy beágyazott rendszerek százai sikeresen hozták át Linux alá a már meglévő kódot (jó példa erre a *Wind River's VxWorks* vagy a *pSOS*, *VRTX*, *Nucleus* és más *RTOS* rendszerek) a módszer gyakorlata azért még nem teljesen kitaposott út.

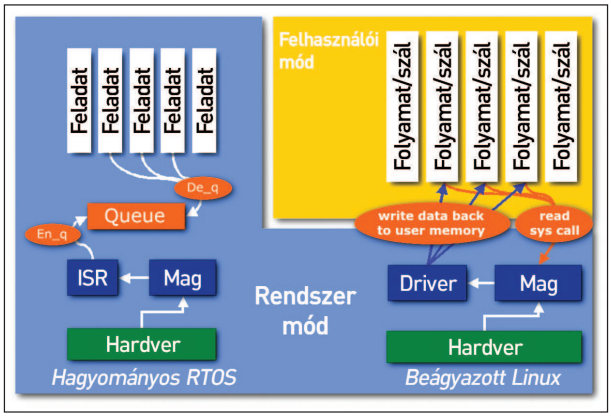
Mind a mai napig az örökölt *RTOS* alkalmazások áthozatalával foglalkozó irodalom nagy része az *RTOS* API-ra, feladatkezelésre és ütemező modellekre és ezek *Linux* felhasználó térbeli megfelelőire koncentrál. Az erősen I/O-függő beágyazott programozási témában legalább ilyen fontos az *RTOS* alkalmazás alkatrész csatoló kódjának a szabványos *Linux* eszközmeghajtó modellhez igazítása.

Cikkünkben a hagyományos beágyazott alkalmazásokban leggyakrabban használt memóriatérkép alapú I/O megközelítéseket szeretnénk sorra venni. A kiszolgáló rutinok (*ISR*) és felhasználói-szálak alkatrész eléréseinek ad hoc szerű felhasználásától kezdve egészen a bizonyos *RTOS* eszköztárakban megtalálható félig-formális meghajtó modellekig jutunk el. Bemutatjuk azokat a heurisztikus és egyéb módszereket melyekkel az *RTOS* kódot szépen formázott *Linux* eszközmeghajtóvá alakíthatjuk. A cikkünk különös figyelmet szentel az *RTOS* kód és a *Linux* memória belapozási eltéréseinek, a sor alapú I/O sémák átírásának, valamint részletesen foglalkozunk az *RTOS* I/O, helyi *Linux* meghajtók és démonok számára használható formára hozásával.

RTOS I/O megoldások

Számos *RTOS* alapú rendszerben alkalmazott I/O technikára leginkább talán a „kötetlen" szó illik. A legtöbb *RTOS* régebbi *MMU* nélküli processzorokra készült, így aztán akkor is figyelmen kívül hagyják a memóriakezelést, ha történetesen van *MMU* és nem tesznek különbséget a logikai és fizikai címzés között. A legtöbb *RTOS* teljes egészében privilegizált módban (rendszer módban) fut, ami látszólag növeli a teljesítmény. Mint ilyen, az *RTOS* alkalmazás és rendszer kód elérheti a gép teljes címterét, a memóriába lapozott eszközöket és I/O utasításokat. Tulajdonképpen meglehetősen nehéz elválasztani egymástól az *RTOS* alkalmazás kódot a meghajtó kódtól, már ahol egyáltalán létezik ilyen megkülönböztetés.

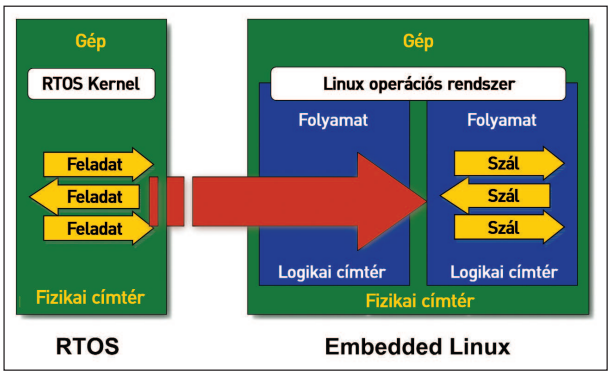
Ez a kötetlen megoldás az I/O megoldások ad hoc jellegű felhasználásához és sok esetben a felismerhető meghajtó modell teljes hiányához vezet. Az ilyen egyenlőség elvű, felosztás nélküli munkakezelés tükrében tanulságos lehet megnézni az *RTOS*-alapú alkalmazásokra vonatkozó néhány kulcselgondolást és gyakorlatot.



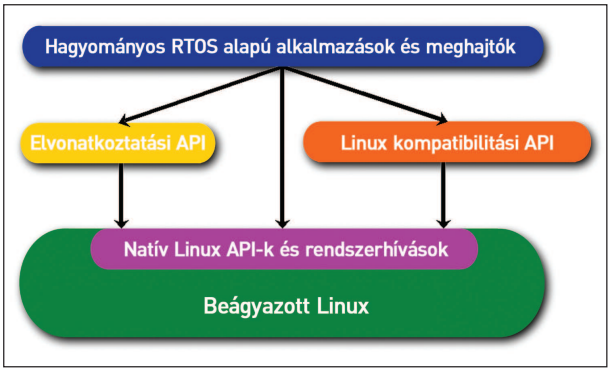
1. ábra Tipikus I/O és adatküldés összehasonlítása hagyományos RTOS és Linux rendszerekben

Sorközi (in-line) memóriába lapozott elérés

Amikor az 1980-as évek közepén megjelentek az üzleti *RTOS* termékek, a legtöbb beágyazott program nagy fő ciklusokat tartalmazott amelyeket az idő-kritikus műveleteket kezelő I/O és *ISR* beszúrások tarkítottak. A fejlesztők első-



2. ábra RTOS feladatok átültetése Linux folyamat alapú szálakká



3. ábra Többágú megközelítés az RTOS kód és API-k Linux átiratához

sorban a párhuzamosság erősítése érdekében terveznek *RTOS* és végrehajtó vezérlést a projektjeikbe, ugyanakkor idegenkedtek bármilyen más szóba kerülő szerkezettől. Így aztán még ha az *RTOS* egyébként lehetővé is tette az I/O formalizmust, a beágyazott programozók inkább a sorközi I/O utasításokat használták:

```
#define DATA_REGISTER   0xF00000F5

char getchar(void) {
  return (*((char *) DATA_REGISTER));
}


void putchar(char c) {
  *((char *) DATA_REGISTER) = c;
}
```

A fegyelmezettebb fejlesztők általában elhatárolták az összes ilyen I/O kódot az eszközfüggetlen kódtól, de rengeteg I/O spagettivel is találkoztam már. Amikor ilyen erőteljes sorközi, memóriába lapozott I/O felhasználással találkoznak, a *Linux* témában zöldfülűnek számító beágyazott fejlesztők gyakran kísértést éreznek, hogy a kódot egy az egyben átvigyék felhasználói térbe és a regisztercímek `#define` megadásait `mmap()` hívásokká alakítsák. Ez a megközelítés bizonyos prototípus megadásoknál jól működik, de nem támogatja a megszakításkezelést, korlátozott valós idejű érzékenységgel rendelkezik, nem igazán biztonságos és üzleti megoldásokban semmiképp sem megfelelő.

RTOS ISR-ek

Linux alatt a megszakítás szolgáltatások kizárólag a rendszermag birodalmába tartoznak. *RTOS* alatt az *ISR* kód szabad formájú és visszatérési módján kívül gyakran semmi nem különbözteti meg az alkalmazás kódtól. Sok *RTOS* ajánl fel olyan rendszerhívást vagy makrót amellyel a kód meghatározhatja saját környezetét. Ilyen például a *Wind River VxWorks* `intContext()` függvénye. Általános megoldás továbbá, hogy az *ISR*-ek szabványos könyvtárakat használnak, ezáltal további újra-belépési és hordozhatósági problémákat vetve fel.
A legtöbb *RTOS* támogatja az *ISR* kódok regisztrációját valamint magára vállalja a megszakítás elbírálást és *ISR* indítást. Bizonyos primitív beágyazott vezérlők ugyanakkor csak az *ISR* kezdőcímek közvetlen illesztését támogatják az alkatrész vektortábláiba. Még ha az írási-olvasási műveleteket sorközi módszerrel a felhasználói térbe helyezzük is, a *Linux ISR* kódnak mindenképpen a rendszermag térbe kell kerülnie.

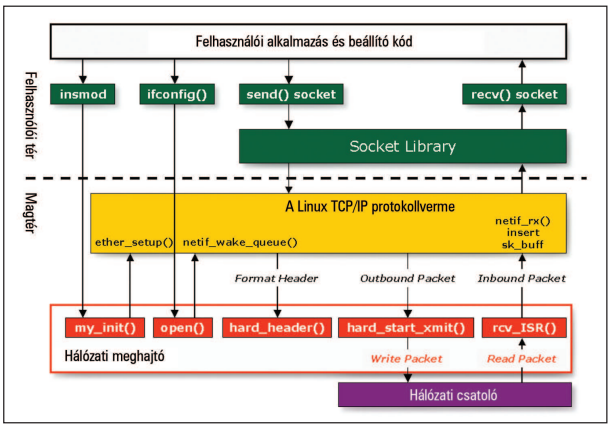
RTOS I/O alrendszerek

A legtöbb *RTOS* testreszabott szabványos C futásidejű könyvtárral rendelkezik (ilyen a *pREPC* a *pSOS* esetében) a *ISV*-khez pedig szelektíven foltozott C könyvtárakat (*libc*) alkalmaz. A *glibc*-vel is ugyanez a helyzet. Így az *RTOS*-nak legalább a szabványos C-stílusú I/O alapelemeit támogatnia kell, ide értve az `open`, `close`, `read`, `write` és `ioctl` rendszerhívásokat. A legtöbb esetben ezek a hívások és hozzátartozóik vékony réteget alkotnak az I/O alapelemek körül. Érdekes módon, mivel a legtöbb *RTOS* nem támogat fájlrendszereket, azok, amelyek Flash vagy forgó médiához szánt fájl absztrakciós réteget is tartalmaznak gyakran teljesen más kódot és/vagy API-t alkalmaznak. Jó példa erre a *pSOS pHILE* rendszere. A *Wind River VxWorks* a legtöbb *RTOS* rendszert lepipálva képességekben gazdag I/O alrendszert is kínál, elsősorban a hálózati csatolófelületek és média kapcsán felmerülő nehézségek kezelésének általánosítása érdekében.
Sok *RTOS* támogatja az alsó fél kezelést, azaz, valamilyen módszert amivel az I/O feldolgozás átvihető a megszakítható és/vagy preemptív környezetbe. Mások nem rendelkeznek ilyesmivel viszont van valamilyen más, hasonló eredményt biztosító mechanizmusuk, például megszakítás egymásba ágyazás.

Tipikus RTOS alkalmazás I/O szerkezete

Az *1. ábra* diagrammja általánosan használt I/O sémát (csak bemenet) és adatközvetítési útvonalat mutat be a vezérlő alkalmazáshoz. A feldolgozás a következő módon zajlik:

- az alkatrész megszakítás kiváltja az *ISR* végrehajtását.
- Az *ISR* elvégzi az alapfeldolgozást és vagy helyben befejezi a bemeneti műveletet vagy az *RTOS*-ra bízza az időzített végrehajtást. Bizonyos esetekben a későbbre halasztott végrehajtást valami olyasmi kezeli, amit Linux alatt felhasználói szálnak itt pedig normál *RTOS* feladatnak nevezünk.
- Ahol és amikor az adat ténylegesen beérkezik (azaz az *ISR* vagy a lehalasztott környezetben) a kész adatok sorba rendeződnek. Az *RTOS ISR* elérheti az alkalmazás sor *API*-t és egyéb *IPC*-ket – lásd az *API* táblát.



4. ábra Linux hálózati meghajtók blokkdiagrammja

- Egy vagy több alkalmazás feladat aztán kiolvassa az üzenetet a sorból és felhasználja az átadott adatokat.

A kimenetet gyakran egyedülálló módszerekkel oldják meg – `awrite()` vagy hasonló rendszerhívások helyett egy vagy több ***RTOS*** alkalmazás-feladat helyez adatokat a sorba.
A sort aztán egy I/O rutin vagy ***ISR*** olvassa be, amely válaszol a küldésre kész megszakításnak, rendszerórának vagy más a sor tartalmára várakozó alkalmazás feladatnak. Ezután az I/O műveletet követlenül végzi, szavazva (polled) vagy DMA-n keresztül.

RTOS I/O hozzárendelés Linux-hoz

A fent bemutatott sor alapú termelő/fogyasztó I/O model csak egy a sok ad hoc megközelítés közül amelyeket az örökölt rendszerek alkalmaznak. Folytassuk ezt az egyszerű példát és nézzünk meg néhány további (újra)megvalósítást a beágyazott ***Linux*** rendszerben.
Azok a fejlesztők akik tartózkodnak a Linux meghajtó tervezés jellemzőinek megismerésétől esetleg akik nagyon sietnek, valószínűleg megpróbálják változatlanul áthozni a felhasználói térbe a sor alapú elképzelést. Ebben a meghajtó beillesztési megoldásban a memóriába lapozott fizikai I/O a felhasználói térben jön létre az `mmap()` által biztosított mutató segítségével:

```
#include <sys/mman.h>

#define REG_SIZE    0x4
/* eszköz regiszter méret*/
#define REG_OFFSET 0xFA400000
/* az eszköz fizikai címe*/

void *mem_ptr;
/* visszahivatkozás a memóriába lapozott
➥eléréshez*/
int fd;

fd=open("/dev/mem",O_RDWR);
/* nyitott fizikai memória (root-nak kell lenni) */

mem_ptr = mmap((void *)0x0,
              ➥REG_AREA_SIZE,
              ➥PROT_READ+PROT_WRITE,
                MAP_SHARED, fd, REG_OFFSET);
/* tényleges  mmap() hívás*/
```

A folyamat alapú felhasználói szál ugyanazt a feladatot látja el, mint amit az ***RTOS***-alapú ***ISR*** vagy halasztott feladat. Ezt követően az ***SVR4 IPC*** `msgsnd()` hívás segítségével a sorba helyezi az üzenetet, ahonnan valamely másik helyi szál vagy folyamat lekérheti azt a `msgrcv()` függvénnyel.
Az ilyen gyors megoldás a prototípusokhoz ugyan elegendő lehet, de fejleszthető kódot készíteni márt komolyabb kihívás. Először szóljunk a megszakítások felhasználói térbe helyezéséről. A ***DOSEMU***-hoz hasonló projektek jelzés alapú I/O megszakításokat használnak a ***SIG***-en keresztül (***silly interrupt generator***, azaz buta megszakításgenerátor) de a felhasználó térbeli megszakítás kezelés meglehetősen lassú – milliszekundumos lappangás a rendszermag alapú ISR pár tíz mikroszekundumos lappangási idejéhez képest. Továbbá a felhasználói környezet ütemezése, még preemptív rendszermag és valós idejű rendszabályok alkalmazása esetén sem garantálnak 100%-ig valós idejű végrehajtást a felhasználói térbeli I/O megszakításoknak.
Mindenképpen javasolt, hogy fogjuk meg a dolog végét és legalább egy egyszerű ***Linux*** meghajtót készítsünk, ami a megszakításokat rendszermag szinten kezeli. Egyszerű karakteres vagy blokkos meghajtó közvetlenül elhelyezheti az alkalmazás adatait a felső félben (top half) vagy elhalasztva a feldolgozást átadhatja azokat egy szálacskának (tasklet), egy rendszermag szálnak vagy a 2.6-os rendszermagban az alsó félben (bottom half) megjelent munkasor mechanizmusnak. Az eszközt egyszerre több alkalmazás szálfolyamat is megnyithatja szinkronizált olvasást végezve rajta, éppen úgy , ahogy az ***RTOS*** alkalmazás végzett szinkronizált sorfogadó hívásokat. Ehhez a megközelítéshez legalább a fogyasztói szál I/O részt újra kell írni, hiszen sor fogadó műveletek helyett most eszközolvasást végzünk.
Amennyiben egyszerűsíteni szeretnénk a beágyazott ***Linuxra*** átvitelt, meghagyhatjuk a sor-alapú sémát és készíthetünk még egy szálat vagy démont amely a frissen elkészített eszköz I/O adataira várakozik. Amikor az adat megérkezett, a szál/démon felébred és az adatot feldolgozó alkalmazás szál vagy folyamat számára elteszi a sorba.

Átalakítási megoldások

Az ***RTOS*** kód beágyazott Linuxhoz alakítása alapelveiben nem különbözik az üzleti alkalmazások átalakításától. Miután az átalakítás logisztikai részével (***make/build*** szkriptek és metódusok, a fordító kompatibilitás, helyek, `include` állományok és így tovább) már végeztünk, a kód szintű átalakítás kihívásai tulajdonképpen az alkalmazás szerkezet és API használat területére szűkülnek.
Témánk további megvitatásához tételezzük fel, hogy az alkalmazás rész (azaz az I/O specifikus részen kívül minden) az ***RTOS***-alapú rendszerből egyetlen ***Linux*** folyamattá alakul át. Az ***RTOS*** feladatok ***Linux*** szálak lesznek a feladatközi ***IPC***-k pedig ***Linux inter-process*** és ***inter-thread*** megfelelőikkel helyettesítjük.

1. táblázat *A Linux és az RTOS-ok előjogokhoz kötött utasításai*

	IPC-k	Szinkronizálás	Feladatkezelés	Névtér
RTOS Alkalmazás	Sorok, jelek, levelesládák kötetlen megosztott memória	Szemafórok, mutexek	Teljes RTOS feladatkezelés Repertoire (Linkelési időben)	Teljes alkalmazás, Könyvtárak és rendszer
RTOS Meghajtó	Sorok, jelek, levelesládák kötetlen megosztott memória	Szemafórok, mutexek	Teljes RTOS feladatkezelés Repertoire	Teljes alkalmazás, Könyvtárak és rendszer (Linkelési időben)
Linux alkalmazás	Sorok, jelek, csővezetékek Intra-Process megosztott memória megosztott rendszer memória	Szemafórok, mutexek	Folyamatok és szálak API-k	Helyi folyamatok, statikus és megosztott könyvtárak
Linux meghajtó (statikus)	Megosztott rendszer memória írható/olvasható folyamatmemória	Rendszermag szemaforok forgózárak (Spinlocks)	Rendszermag szálak, taskletek	Teljes rendszermag
Linux modul (dinamikus)	Megosztott rendszer memória írható/olvasható folyamatmemória	Rendszermag szemaforok forgózárak (Spinlocks)	Rendszermag szálak, taskletek	Modul-szintű és exportált rendszermag szimbólumok

Az átírás alapjait nem nehéz megérteni, a buktató a részletekben rejlik. A leglényegesebb részlet pedig mind között az ***RTOS API*** használata, illetve az, hogyan tudjuk megvalósítani ***Linux*** szerkezetekkel.

Amennyiben a projektünk nem időhöz kötött és célunk hordozható kódot készíteni későbbi projektfejlesztésekhez, akkor érdemes egy kis időt tölteni a jelenlegi ***RTOS*** alkalmazás elemzésével és megvizsgálni miképpen illeszkedik a Linux elképzelésekhez. Az ***RTOS*** alkalmazás kód esetében elgondolkozhatunk rajta, át tudjuk-e Linux folyamat-alapú szálakká alakítani az ***RTOS*** feladatokat egy az egyben avagy inkább a teljes ***RTOS*** alkalmazást több ***Linux*** folyamatra bontjuk. Ettől a döntéstől függően meg kell vizsgálnunk a felhasznált ***RTOS IPC***-ket, hogy a helyes ***intra-process*** vagy ***inter-process*** hatáskört válaszuk.

Meghajtó szinten egyértelmű, hogy valamennyi sorközi ***RTOS*** kódot megfelelő meghajtókká kell alakítani. Amennyiben az örökölt alkalmazásunk már eleve szépen felosztott, akár ***RTOS*** I/O ***API***-kat használ, akár külön rétegekbe szedték szét, feladatunk jelentősen egyszerűsödik. Ha viszont az ad hoc I/O kód az egész örökölt kódbázisunkban szanaszét szórva található saját magunknak kell kiköveznünk az utat. Azok a fejlesztők, akik sietősen szeretnék kiszedni az örökölt ***RTOS*** kódot vagy akik prototípust akarnak összekalapálni, valószínűbb, hogy megpróbálják a lehető legtöbb ***RTOS*** API-t helyben ***Linux*** megfelelővel kiváltani. Az elemeket általában, akárcsak az egyedi ***API***-kat, ***IPC***-ket és rendszer adattípusokat, majdnem teljesen átlátszóan viszik át. Mások a `#define` újradefiniálásban és makrókban hisznek. A többiek újrakódolják a részeket, ideális esetben egy elvonatkoztatási réteg részeként.

Az ***API***-alapú átalakítást érdemes az emulációs könyvtárakkal érdemes kezdeni, amelyek sok beágyazott ***Linux*** terjesztésnek részét képezik (például a ***MontaVista*** könyvtárai a ***Wind River VxWorks*** és ***pSOS*** rendszerekhez) avagy olyan harmadik féltől származó ***API***-átírat csomagokat használnak, mint amilyet például a ***MapuSoft*** kínál.

A legtöbb projekt hibrid megközelítést alkalmazza: az összes egyedi vagy könnyen megvalósítható API-t áthozza, újraszerkesztik azokat a részeket, ahol az nem okoz lassulást majd csapd-le-a-vakondokot módszerrel addig foltozzák a maradék kódot amíg le nem fordul és fut.

A rendszermag és a felhasználói tér elérhető API felületei

Akár a teljes újratervezést, akár a gyors de felületes ***API*** megközelítést választjuk, az ***RTOS*** alkalmazásunkat és az I/O kódot szét kell osztanunk, hogy az illeszkedjen a ***Linux*** rendszermag és felhasználói tér elképzeléshez. Az *1. táblázat* bemutatja a ***Linux*** mennyivel szigorúbb az előjogot igénylő utasítások terén mint az örökölt ***RTOS***. Segítségével könnyebben elvégezhetjük a felosztási rész.

Az 1. táblában két pontra különösen felhívnám a fegyelmet:

- az ***RTOS***-ok egyenlőség pártiak, az alkalmazásnak és az I/O kódnak is lehetővé teszi, hogy bármilyen címhez hozzáférjen és szinte bármilyen tevékenységet végezzen, a ***Linux*** ugyanakkor sokkal hierarchikusabb és kötöttebb.
- Az örökölt ***RTOS*** kód a rendszer valamennyi belépési pontját és szimbólumát láthatja, legalább linkeléskor, ugyanakkor a ***Linux*** felhasználói kód a rendszermag kódtól és annak névterétől teljesen elhatárolva készül.

A ***Linux*** előjoghoz kötött eléréshierarchiájának eredményeképpen általában csak a rendszermag kód (meghajtók) érhetik el a fizikai memóriát. Ha egy felhasználói kód ilyesmit szeretne, annak ***root*** jogon kell futnia. Általában véve a felhasználói tér kódja elhatárolódik a ***Linux*** rendszermag kódtól és csak azokat a szándékosan exportált szimbólumokat éri el, amelyek a ***/proc/ksyms*** alatt látszanak. Továbbá a rendszermag hívásai nem közvetlenül történnek, hanem a felhasználói könyvtár kódon keresztül. Ez a fajta szétválasztás szándékos, célja a ***Linux*** biztonságának és stabilitásának növelése. Amikor meghajtót írunk, minden éppen fordítva van. A statikusan meghajtók a teljes rendszermag névterét látják, nem csak az exportokat, ugyanakkor semmilyen rálátásuk nincs

a felhasználói tér folyamat alapú szimbólumaira és belépési pontjaira. Továbbá, ha a meghajtónkat futásidőben betölthető modulba csomagoljuk, programunk csak azokat a csatolófelületeket befolyásolhatja, amelyeket az `*EXPORT_SYMBOL*` makró segítségével kifejezetten exportáltunk a rendszermagban.

Hálózati meghajtók átírása

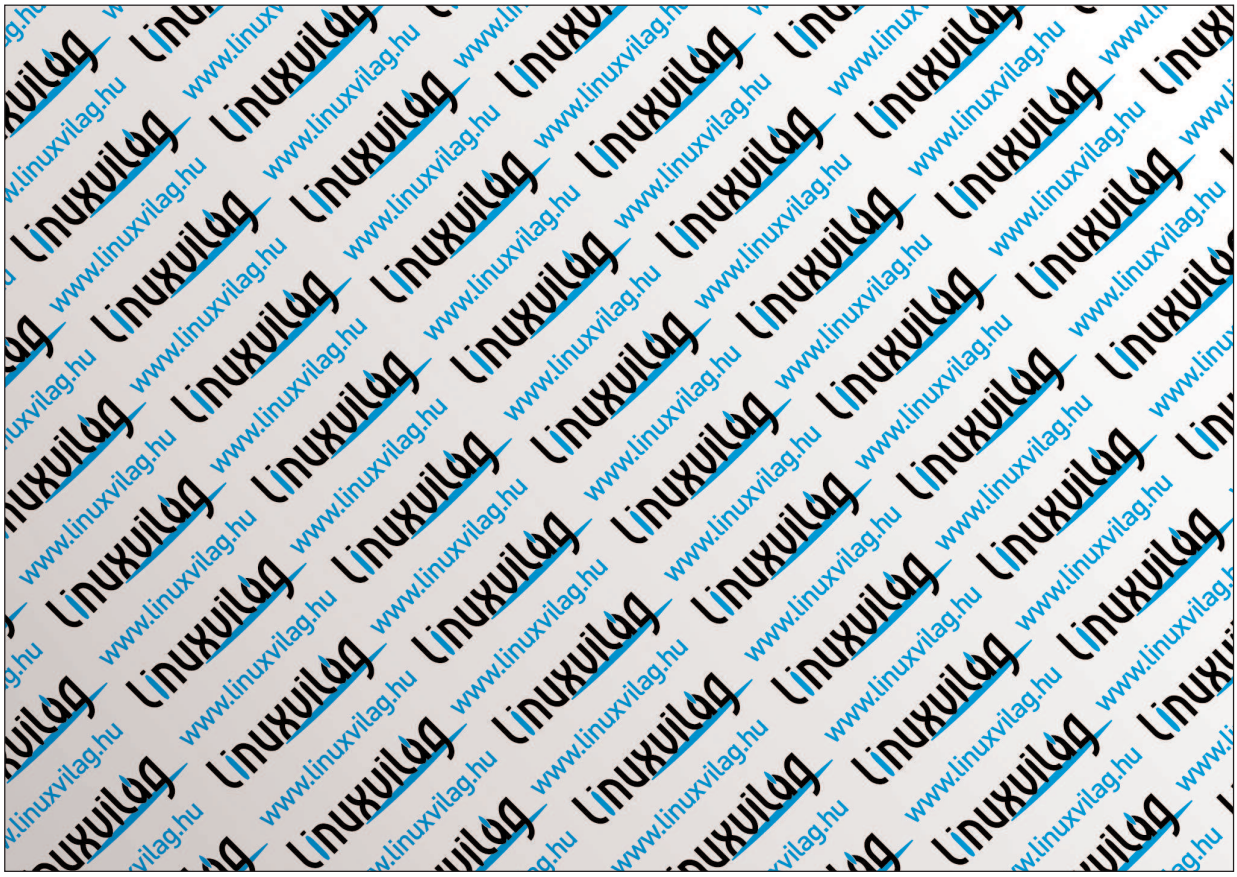
Mint fentebb említettük, karakteres és blokkos eszközök átírása ***Linux*** alá időigényes, de viszonylag magától értetődő folyamat. A hálózati meghajtók átírása viszont már ijesztőbbnek tűnik. Ne feledjük, hogy a ***Linux*** tulajdonképpen a ***TCP/IP***-vel együtt nőtt fel, sok 1990-es évek végén készült ***RTOS*** már tartalmazott hálózati kódot. Így aztán az örökölt hálózatkezelés sokszor csak a képességek vázát tartalmazza, azaz csak egyetlen folyamatot vagy példányt kezel egyetlen kapun avagy csak a fizikai csatolófelületet kezeli egyetlen hálózati médiumon. Néhány esetben a hálózati szerkezetet utólag csiszolgatták. Ez történt például a ***Wind River VxWorks MUX*** kódjával, hogy támogasson több csatolófelületet és fizikai kapcsolattípust. A rossz hír, hogy valószínűleg újra kell írnunk elég sok, ha nem az összes meglévő hálózati csatolófelületet. A jó hír, hogy a ***Linux*** alatti újra felosztás nem olyan nehéz és példának tucatnyi nyílt forrású hálózati kód áll rendelkezésünkre. Átalakítási munkánk feladata, hogy a *4. ábra* alján található részeket megfelelő csomag formázó és csatolófelület kóddal töltsük fel.
Hálózati meghajtók készítése nem kezdőknek való feladat. Mivel azonban sok ***RTOS*** hálózati meghajtó létező ***GPL Linux*** csatolófelületek forrásából készült, elképzelhető, hogy maga a kód segíti elő törekvéseinket. Továbbá, az integrátorok és tanácsadók elég nagy és egyre bővülő közössége odafigyel és üzletet lát a ***Linux*** átalakításokat végző beágyazott fejlesztők részére nyújtott segítségben. Az áraik is elfogadhatóak.

Összefoglalás

Cikkünk célja, hogy a beágyazott fejlesztőknek egy kis rálátást nyújtson, milyen kihívások és előnyök várhatóak amennyiben teljes programkészletüket ***Linux*** alá viszik át az eredeti ***RTOS*** rendszerről. Persze körülbelül 2.800 szó túl kevés rá, hogy igazán elmerüljünk a meghajtóátírás (meghajtó API-k és buszfelületek címfordítás és egyebek) részleteiben, de a seregnyi meglévő ***GPL*** meghajtó kód egyaránt szolgál kiváló dokumentációként és sablonként átültetési erőfeszítéseink során. Az itt bemutatott irányelvek csapatunkat kívánják segíteni az ***RTOS*** kód linuxos átírásában valamint ötletekkel támogatni az eredeti kód beágyazott ***Linux*** rendszerekhez leginkább illeszkedő újrafelosztását.

Linux Journal 2004. október, 126. szám

E cikk írásakor ***Bill Weinberg*** a Strategic Marketing and Technology Evangelist igazgatójaként a MontaVista és beágyazott Linux fejlesztése terén szerzett több mint 17 évnyi ipari tapasztalat tudását összegezte. Kiterjedt tapasztalatokkal rendelkezik a beágyazott valós idejű rendszerek terén és szakértőnek számít az operációs rendszerek, eszközök, program licencek és programkészítés témákban.



A HTB várakozásisor-kezelési elv elemzése

Vajon képes a Linux úgy garantálni a szolgáltatásminőséget, hogy egyszerre álljon rendelkezésünkre a kívánt sávszélesség, a megadott határt mégse lépjük túl? Kipróbáltuk.

A ***HTB (Hierarchical Token Buckets, hierarchikus zseton vödrök)*** várakozásisor-kezelő elv – a linuxos forgalomszabályozó megoldások egyik eleme – olyan eszköz, mely egyrészt ***szolgáltatásminőségi (Quality of Service, QoS)*** funkciókat biztosít, másrészt alkalmas a ***TCP*** alapú forgalom kifinomult szabályozására. Írásomban áttekintem a sorkezelő elv elemeit, valamint több előzetes teljesítményteszt eredményeit is ismertetem. Ezek elvégzéséhez különféle linuxos környezeteket állítottunk össze, a forgalomkeltést pedig egy Ixia készülékre bíztuk. A próbák során kimutattuk, hogy az átviteli sebesség szabályzásának pontossága befolyásolható, illetve a sávszélesség körülbelül egy 2 Mbps-os tartományon belül szabályozható. A próbák igazolták a ***HTB*** sorkezelő algoritmusok teljesítményét és pontosságát, valamint rámutattak bizonyos, a forgalomkezelés javítására alkalmas módszerekre.

A forgalomszabályozási eljárások akkor jutnak szerephez, amikor az adott ***IP***-csomag már bekerült a továbbításra váró csomagok adott kimenő felülethez rendelt várakozási sorába, ám az illesztőprogram még nem kezdte meg tényleges továbbítását. Az ***1. ábrán*** látható, hogy a forgalomszabályozási döntések meghozatalának helyszíne hogyan viszonyul a fizikai Ethernet összeköttetésen való továbbításhoz és a szállítási rétegbeli protokollokhoz, mint az ***UDP*** és a ***TCP***.

A ***Linux*** rendszermag ***Alexey Kuznetsov*** által megvalósított forgalomszabályozó szolgáltatása négy fő összetevőből áll: várakozásisor-kezelő elvek, szolgáltatási osztályok, szűrők és házirendkezelés.

A várakozásisor-kezelő elvek a sorba állított ***IP***-csomagok kezelésének módját megadó szoftveres eljárások. Minden hálózati készülék ismer valamilyen sorkezelő elvet, ezek közül a legelterjedtebb talán a ***FIFO (First In, First Out, elsőként érkező elsőként távozik)*** algoritmusra alapuló elv, melynél a csomagok érkezésük sorrendjében kerülnek be a várakozási sorba, mégpedig olyan sebességgel, amilyennel a sorhoz tartozó készülék képes elküldeni őket. A ***Linux*** jelenleg is számos sorkezelő elvet támogat, és újabbak hozzáadására is biztosít lehetőséget.

A sorkezelő algoritmusok részletes leírását az interneten ***„Iproute2+tc Notes”*** (lásd a forrásokat) címmel lehet megtalálni. A ***HTB*** elv a hozzárendelt szolgáltatási osztályok soraiba helyezett csomagok kezelésére a ***TBF*** algoritmust alkalmazza. A ***TBF*** algoritmus forgalmi házirendkezelési és forgalomformálási (traffic-shaping) képességekkel rendelkezik. A ***TBF*** algoritmus részletes leírása a ***Cisco IOS Quality of Service Solutions Configuration Guide*** útmutatójában (lásd: ***„Policing and Shaping Overview”***, A házirendkezelés és a forgalomformálás áttekintése) szerepel.

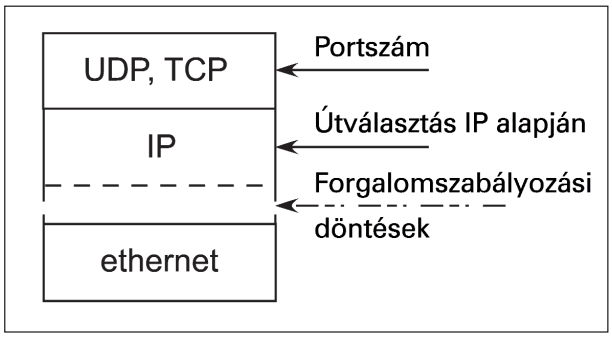
Egy-egy szolgáltatási osztály bizonyos házirendszabályokat határoz meg, ilyen a maximális sávszélesség vagy a maximális löket, és a sorkezelő elvet alkalmazza ezen szabályok betartatására. Egy-egy sorkezelő elv és osztály összetartoznak. Az adott osztály által előírt szabályokat egy előre megadott várakozási sorral kell összerendelni. A legtöbb esetben minden osztálynak saját sorkezelő elve van, de előfordulhat, hogy több osztály is osztozik ugyanazon a várakozási soron. A sorkezelő megoldásoknál az adott osztályhoz tartozó házirend összetevők általában eldobják a megadott határértéken túli csomagokat (lásd: ***„Policing and Shaping Overview”***).

A szűrők a sorkezelő elv által alkalmazott szabályokat adják meg. A sorkezelő elv ezen szabályok alapján dönti el, hogy melyik osztályhoz kell rendelnie a bejövő csomagokat. Minden szűrőnek megadott fontossága, prioritása van. A szűrők fontosságuk szerint növekvő sorrendbe vannak rendezve. Amikor egy sorkezelő elv kap egy csomagot sorba állításra, akkor megpróbálja egyeztetni azt a megadott szűrők valamelyikével. Az egyezés keresése a lista szűrőinek megvizsgálásával történik, kezdve a legnagyobb fontosságúval. Minden osztályhoz vagy sorkezelő elvhez egy vagy több szűrő tartozhat.

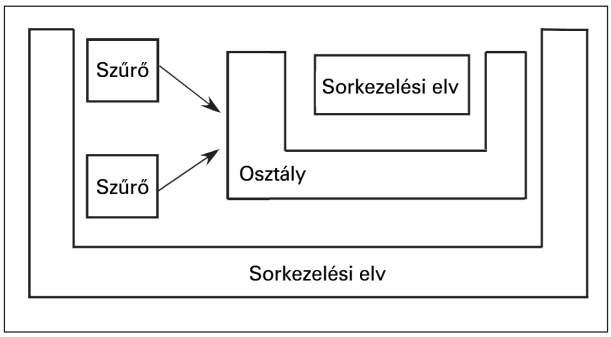
A házirendkezelő összetevők biztosítják, hogy a forgalom nagysága ne haladja meg a megadott sávszélességet. A házirendkezelő döntések a szűrőkön és az osztály alapon megadott szabályokon alapulnak. A ***Linux*** forgalomszabályozó alrendszerének összetevői között fennálló kapcsolatokat a ***2. ábra*** tartalmazza.

A TC segédprogram és a HTB definíciók

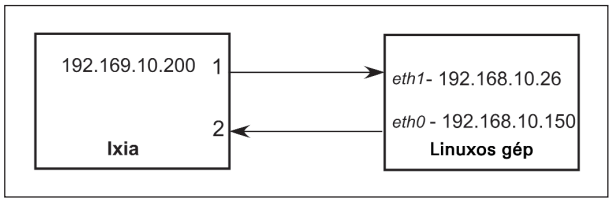
A ***TC*** egy felhasználói program, segítségével várakozási sorokat, osztályokat és szűrőket lehet létrehozni, illetve hozzá lehet ezeket rendelni a megfelelő kimenő felülethez (lásd: ***„tc-Linux QoS control tool”*** a források között). A szűrőket az irányítótábla, ***u32*** osztályozók és ***TOS*** osztályozók alapján lehet összeállítani. A program ***netlink*** foglalatokon



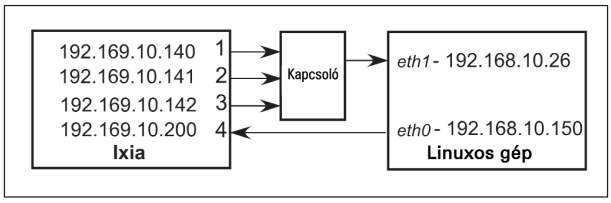
1. ábra A rendszermag azt követően hozza meg forgalomszabályozási döntéseit, hogy a csomag bekerült a küldési várakozási sorba



2. ábra A Linux forgalomszabályzó alrendszerében sorkezelő elveket, osztályokat, szűrőket és házirendkezelést találunk



3. ábra Az 1. tesztmodell összeállítása



4. ábra Az 2. tesztmodell összeállítása

keresztül tartja a kapcsolatot a rendszermag hálózatkezelő alrendszerének eljárásaival. Az ***1. táblázatban*** a három fő műveletet és a megfelelő *TC* parancsokat foglaltam össze. A *TC* használatáról a ***HTB Linux Queuing Discipline User Guide*** útmutató tartalmaz részletesebb tájékoztatást.
A *HTB* használata az egyik mód arra, hogy kézben tartsuk az egyes összeköttetések kimenő sávszélességének használatát.

1. táblázat *A TC segédprogrammal elvégezhető műveletek és a megfelelő parancsok*

TC művelet	Parancs
tc qdisc	Sorkezelő elv létrehozása
tc filter	Szűrő létrehozása
tc class	Osztály létrehozása

A *HTB* használatához azt osztályként és a sorkezelési elv típusaként kell megadni. A *HTB* a forgalmat a *TBF* algoritmus segítségével alakítja, mely független az alsóbb réteg sávszélességétől. Kizárólag a gyökér sorkezelő elvet szabad *HTB* típusként megadni, az összes többi osztálypéldány a *FIFO* sort (alapértelmezett) használja. A sorkezelő folyamat mindig a gyökér szinten indul, majd a szabályok alapján dönti el, hogy melyik osztálynak kell megkapnia az adatokat. Az osztályfa bejárása egészen addig folytatódik, míg egyező levélosztályt nem sikerül találni (lásd: *„Hierarchical Token Bucket Theory”*).

Tesztelés

A *HTB* pontosságát és teljesítményét a következő hálózati készülékek segítségével próbáltuk ki: egy darab ***Ixia 400*** forgalomkeltő készülék, egy darab ***10/100 Mbps Ethernet*** terhelő modullal ***(LM100TX3)***, továbbá egy darab ***Pentium 4 PC*** (1 GB RAM, 70 GB-os merevlemez), ezen a Linux rendszermag 2.6.5-ös változata futott. Két próbamodellt terveztünk, egyet a házirendkezelés pontosságának, a másikat a sávszélesség megosztásának ellenőrzésére.
Az első modellt (***3. ábra***) meghatározott osztály házirendkezelési pontosságának ellenőrzésére használtuk. Az ***Ixia 1***-es kapuján keltett forgalom egy vagy több folyamot alkotott, célállomása a ***192.168.10.200 IP***-cím volt. A linuxos gép a csomagokat egy statikus útvonallal az eth0 felületre irányította, majd visszaküldte őket az ***Ixia 2***-es kapujára.
A forgalomszabályozási előírásokat az eth0 felületre léptettük életbe. Az elemzéseket teljes egészében az ***Ixia 2***-es kapuján kapott forgalom alapján végeztük.
A második modellt (***4. ábra***) annak vizsgálatára használtuk, hogy két, azonos osztályba tartozó adatfolyam sávszélességének megosztása hogyan történik. Ebben az esetben további két ***Ixia*** kaput vettünk igénybe az adatok küldésére.
Az ***Ixia*** 1., 2. és 3. kapuja – egy-egy adatfolyamot használva – egyaránt mesterségesen keltett forgalmat továbbított a ***192.168.10.200***-as *IP*-címre. A linuxos gép ezeket a csomagokat egy statikus útvonal alapján az eth0 felületre irányította, majd visszaküldte őket az ***Ixia*** 2-es kapujára. A forgalomszabályozási szabályok ebben az esetben is az eth0 felületre vonatkoztak. Minden elemzést az ***Ixia*** 2-es kapujára beérkezett forgalom alapján végeztünk.

Az Ixia készülék beállításai és korlátai

A küldő kapuk minden próbánál összefüggő csomaglöketeket küldtek, megadott sávszélességgel. Az ***Ixia 10/100 Mbps*** sebességű ***Ethernet (LM100TX3 modellszámú)*** terhelő modulja négy különböző kapuval rendelkezik, ezek mindegyike legfeljebb 100 Mbps sebességű küldésre képes. Az ***Ixia*** terhelőmodulja képes volt ugyan arra, hogy egy-egy kapun több adatfolyamot állítson elő, de nem tudta összekeverni

egymással az adatfolyamokat, és egyszerre csak egy adatfolyamot szolgáltatott. A korlátozás oka, hogy ütemezője körbeforgó elven működik. Először elküldi az X adatfolyam egy bájtlöketét, majd továbblép a következő adatfolyamra, és az Y adatfolyamon belül is küld egy löketet.
Ha egy adatfolyamból meghatározott sávszélességet akarunk előállítani, mely ráadásul az adott kapuhoz hozzárendelt adatfolyamcsoport tagja, akkor finomhangolnunk kell az Ixia bizonyos beállításait. Ezek a beállítások – illetve szerepük – következők:

- ***Burst (löket)***: az egyes folyamokban a következő folyamra való továbblépés előtt elküldött csomagok száma
- ***Packet size (csomagméret)***: a folyamot alkotó egyes csomagok mérete
- ***Total bandwidth (teljes sávszélesség)***: az összes adatfolyam által felhasznált sávszélesség

Az ***Ixia*** beállításait a ***2. táblázatban*** foglaltuk össze.
A cél az volt, hogy meghatározzuk, mi az a löketméret, amelynél az egyes adatfolyamokhoz kívánt sávszélességet el tudjuk érni. Mivel mindhárom adatfolyam ugyanazt a fizikai vonalat használta, az adatok továbbításának módja az ***5. ábrán*** láthatóhoz volt hasonló.
A beállítások közötti összefüggéseket az alábbi egyenletekkel írhatjuk le:

$$T_c = \mathrm{Sum}(B_{s\text{-}i} * 8 * P_{s\text{-}i}) / T_b$$
$$N_c = 1/T_c$$
$$B_{n\text{-}i} = N_c * P_{s\text{-}i} * 8 * B_{s\text{-}i}$$

Az egyenletekben használt változók jelentését a ***3. táblázat*** tartalmazza.
Feltételezve, hogy a csomagméret minden adatfolyam esetében azonos, ahogy a példában is, csak a löketméret meghatározása marad hátra.
Mivel minden adatfolyam ugyanazon a sávszélességen osztozik, az elvárt löketértékeket az elvárt sávszélességek közötti arányok vizsgálatával kaphatjuk meg, a $B^{s\text{-}i} = B^{n\text{-}i}$ egyenlet alapján. Ez az érték meglehetősen nagy is lehet, szükség esetén osztásokkal hozhatjuk kezelhetőbb formára. Ha különböző csomagméreteket akarunk hozzárendelni az egyes adatfolyamokhoz, akkor a löketméretek szükség szerint ismételt módosításával kaphatjuk meg az egyes adatfolyamokban a kívánt sávszélességeket. Táblázatkezelő program segítségével egyszerűbben, könnyebben végezhetjük el a több sávszélességre is kiterjedő számításokat.

Tesztek és eredmények

A HTB beállításainak megadásakor a következő tc class parancsokat használtuk az elvárt eredmények eléréséhez:

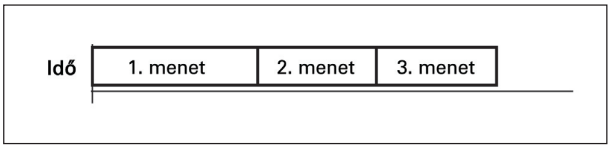
- *rate* = adott osztály legfeljebb ennyi sávszélességet használhat el úgy, hogy nem más osztályoktól veszi azt kölcsön

2. táblázat ***Az Ixia készülék beállításai***

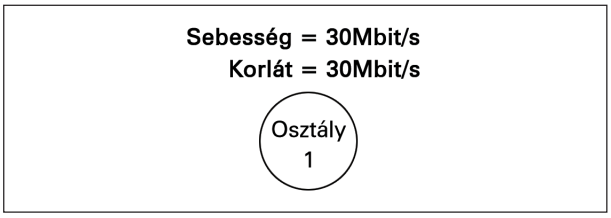
Adatfolyam	Mesterségesen keltett sávszélesség	Csomagméret	Löketméret
1	15 Mbps	512 B	150
2	10 Mbps	512 B	100
3	2 Mbps	512 B	20
Összes	27 Mbps	–	–

3. táblázat ***A beállítások összefüggéseit jellemző egyenletekben használt változók***

Beállítás	Jelentés
T_c	Az 1-i sorszámú löketek elküldéséhez szükséges idő összege másodpercben mérve ($T_c1 + T_c2 + T_c3+...$)
$B_{s\text{-}i}$	Az i adatfolyam löketeit alkotó csomagok száma
$P_{s\text{-}i}$	Az i adatfolyamban elküldött csomag mérete
T_b	Az összes adatfolyam által lefoglalt sávszélesség (bit/másodperc)
N_c	A T_c löketek másodpercenkénti száma
$B_{n\text{-}i}$	Az i adatfolyam elvárt sávszélessége (bit/másodperc)



5. ábra Az adatok áramlása az átviteli vonalon az Ixia készülék és a tesztelt Linux rendszer között



6. ábra 1. teszt – egy-egy be- és kimenő adatfolyam

- *ceiling* = adott osztály legfeljebb ennyi sávszélességet használhat el; így szabályozható, hogy az osztály mekkora sávszélességet vehet kölcsön másoktól
- *burst* = maximális sebességgel legfeljebb ennyi adatot lehet elküldeni, mielőtt a következő osztály kiszolgálására lépnénk tovább
- *cburst* = az átviteli közeg sebességével legfeljebb ennyi adatot lehet elküldeni, mielőtt a következő osztály kiszolgálására lépnénk tovább

4. táblázat *Az 1. tesztmodellel kapott eredmények*

Burst (bájt)	Cburst (bájt)	Csomagméret (bájt)	Bejövő sávszélesség (Mbps)	Kimenő sávszélesség (Mbps)
Alapértelmezett	Alapértelmezett	128	40	33,5
Alapértelmezett	Alapértelmezett	64	40	22
Alapértelmezett	Alapértelmezett	64	32 (Max)	29,2
15 k	15 k	64	32 (Max)	30
15 k	15 k	512	32 & 50 & 70	25,3
15 k	15 k	1500	32 & 50 & 70	25,2
18 k	18 k	64	32 (Max)	29,2
18 k	18 k	512	32 & 50 & 70	30,26
18 k	18 k	1500	32 & 50 & 70	29,29

5. táblázat *A Burst/Cburst és Rate értékek közötti kapcsolat*

Burst (bájt)	Cburst (bájt)	Csomagméret (bájt)	Bejövő sávszélesség (Mbps)	Kimenő sávszélesség (Mbps)	Hozzárendelt sebesség (Mbps)
9 k	9 k	64	32	17,5	15
9 k	9 k	512	32	15,12	15
9 k	9 k	1500	32	15,28	15
4,8 k	4,8 k	64	32	8,96	8
4,8 k	4,8 k	512	32	8,176	8
4,8 k	4,8 k	1500	32	8	8
3 k	3 k	64	32	17,5	15
3 k	3 k	512	32	15,12	15
3 k	3 k	1500	32	15,28	15

Az internetes világ forgalmának túlnyomó része ***TCP*** alapú, ezért a próbák során is a datagramokra jellemző – 64 bájtos (TCP Ack), 512 bájtos (FTP) és 1500 bájtos – csomagméreteket választottunk.

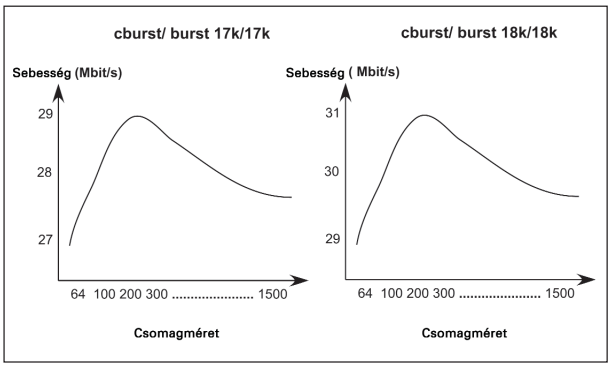
Az első tesztmodell

A ***4. táblázatban*** foglalt eredmények alapján a következőket jelenthetjük ki:

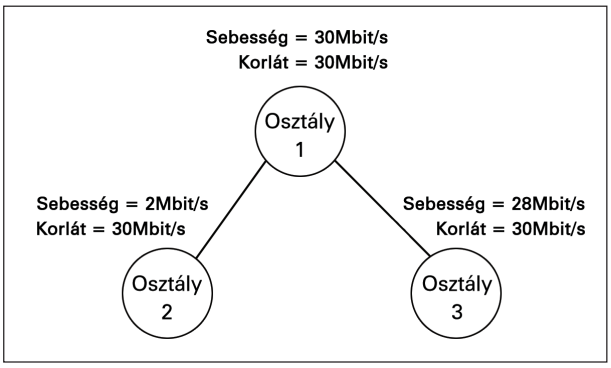
- Egy linuxos gép 64 bájtos csomagokból álló, összefüggő adatfolyamok esetén körülbelül 34 Mbps sebességgel képes továbbítani (egyik interfészen fogadni, a másikon küldeni) a forgalmat.
- A legjobb átlagos szabályzási pontosságot 18 k / 18 k burst/cburst érték mellett kapjuk.
- A löket (burst) érték és az elvárt sebességérték között lineáris kapcsolat van, ez a tesztek során egyértelművé válik.
- A kimenő felületre erőltetett sávszélesség nem befolyásolja az eredmények pontosságát.

A ***7. ábra*** a csomagméret és a kimenő sávszélesség közötti, különféle csomagméretekkel elvégzett tesztek során kapott eredmények alapján megállapított kapcsolatot szemlélteti. A 4. táblázatban és a 7. ábrán szereplő eredményekből két tanulságot vonhatunk le: egyrészt az átvitel szabályzásának pontosságát a ***cburst/burst*** értékek módosításával lehet befolyásolni, másrészt a pontossági sávszélesség-tartomány 2 Mbps széles, amennyiben a csomagméret 64 és 1500 bájt közötti. Azt, hogy a ***burst/cbursts*** értékek és az átviteli sávszélesség (rate) között lineáris összefüggés van, többféle ***burst*** és ***cburst*** érték vizsgálatával igazoltuk. Az ***5. táblázat*** a próbák során kapott adatminták fontosabbjait tartalmazza. A kiindulási érték 18 k / 18 k volt. A ***burst/cburst*** értékeket a ***cburst/burst*** (Kb) = 18/(30M/hozzárendelt sebesség) képlet alapján kaptuk. Az 5. táblázatban szereplő eredmények szerint a ***cburst/burst*** értékeket dinamikusan lehet megadni a sebességértékhez, mint amikor lineáris kapcsolatot feltételezünk.

A ***6. táblázat*** az egyszintű öröklődésre mutat példát. A 2. és a 3. osztály örökli az 1. osztály sebességkorlátját (30 Mbps).



7. ábra A csomagméret és a kimenő sávszélesség kapcsolatának grafikus elemzése



8. ábra 2. teszt – három bejövő és egy kimenő adatfolyam

6. táblázat *A 2. teszt eredményei*

Adatfolyam	Burst (bájt)	Cburst (bájt)	Csomagméret (bájt)	Bejövő sávszélesség (Mbps)	Kimenő sávszélesség (Mbps)	Osztály
1	17 k	17 k	64	15	12,7	3
2	17 k	17 k	512	20	17,1	3
3	1 k	1 k	512	4	2,01	2
Összes	18 k	18 k	–	39	31,8	–

7. táblázat *A 3. teszt eredményei*

Adatfolyam	Burst (bájt)	Cburst (bájt)	Csomagméret (bájt)	Bejövő sávszélesség (Mbps)	Kimenő sávszélesség (Mbps)	Osztály
1	1 k	1 k	512	5	2,04	2
2	6 k	6 k	6	15	11,326	4
3	3 k	3 k	64	10	5,67	5
4	7,8 k	7,8 k	512	20	13,02	6
Összes	18 k	18 k	–	50	32,05	–

Ennél a tesztnél a gyermekosztályok felső sebességkorlátja megegyezik a szülő korlátjával, vagyis a 2. és a 3. osztály legfeljebb 30 Mbps sávszélességet vehet kölcsön. A többi osztályhoz tartozó ***cburst/burst*** értékeket lineáris összefüggést feltételezve, az elvárt sávszélességek alapján számítottuk ki.

A 6. táblázat azt szemlélteti, hogy egyszintű öröklésnél hogyan jellemezhető a lineáris összefüggés. A teszt során a bejövő adatfolyam 39 Mbps sávszélességű, folyamatos forgalmat ad, a teljes kimenő sávszélesség pedig 31,8 Mbps.

A *7. táblázat* a kétszintű öröklést szemlélteti. A 2. és a 3. osztály örökli az 1. osztály sebességkorlátját (30 Mbps). A 4., az 5. és a 6. osztályok a 3. osztály sebességkorlátját öröklik (28 Mbps), és saját sebességhatáraik szerint osztoznak rajta. A tesztben a gyermekosztályok felső sebességkorlátja megegyezik a szülő korlátjával, vagyis a 4., az 5. és a 6. osztály legfeljebb 28 Mbps sávszélességet vehet kölcsön. A többi osztályhoz tartozó ***cburst/burst*** értékek kiszámítása lineáris összefüggést feltételezve, az elvárt sávszélességek alapján történt.

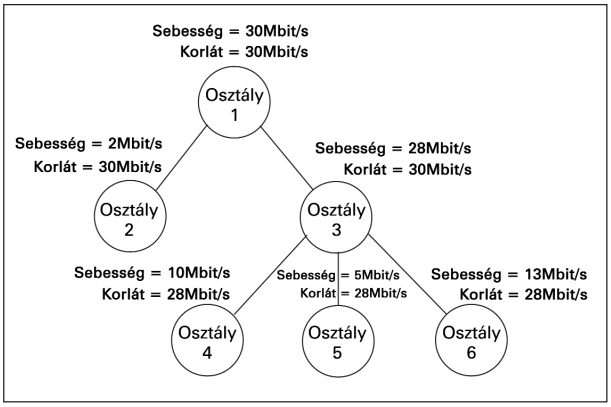
A 7. táblázatban foglalt eredmények alapján megállapíthatjuk, hogy a lineáris kapcsolat kétszintű öröklésnél is fennáll. A teszt során a bejövő kapu 50 Mbps sávszélességű, folyamatos forgalmat fogad, a teljes kimenő sávszélesség pedig 32,05 Mbps.

A második tesztmodell

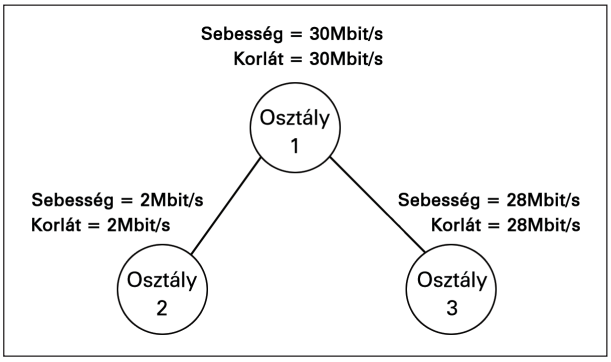
Ahogy a *8. ábrán* is követhető, a sávszélesség egyenlően oszlik meg az adatfolyamok között, amennyiben ezek azonos számú bájtot küldenek el, illetve azonos osztályhoz tartoznak. Egy másik kísérlet, amelynél az 1. adatfolyam bejövő sávszélessége nagyobb volt, mint a 2. és a 3. adatfolyamé, kimutatta, hogy az 1. adatfolyamnak a kimenő sávszélessége is nagyobb volt, mint a 2. és a 3. adatfolyamé. Az eredményekből arra következtethetünk, hogy ha több ada-

8. táblázat *Az 1. teszt eredményei*

Adatfolyam	Burst (bájt)	Cburst (bájt)	Csomagméret (bájt)	Bejövő sávszélesség (Mbps)	Kimenő sávszélesség (Mbps)	Osztály
1	1 k	1 k	512	5	0,650	2
2	1 k	1 k	512	5	0,600	2
3	1 k	1 k	512	5	0,568	2
Összes	18 k	18 k	–	15	1,818	–



9. ábra 3. teszt – négy bejövő és egy kimenő adatfolyam



10. ábra 1. teszt – három bejövő, egy kimenő adatfolyam

tot küldünk egy meghatározott adatfolyamban, akkor az az azonos osztályba tartozó egyéb folyamokhoz képest több csomag továbbítására lesz képes.

Összefoglalás

Az ismertetett kísérletek elvégzése az egyik lehetőség a *HTB* pontosságának és teljesítményének vizsgálatára. Bár a folyamatosan érkező, adott forgalmat jelentő löketek nem feltétlenül jellemzőek a valós életbeli adatforgalomra, vizsgálatukkal mégis megkaphatjuk a kiindulási pontokat a *HTB* osztályok és jellemzőik megadásához.
A kísérletek eredményeit a következő kijelentésekkel összegezhetjük:

- Egy linuxos gép 64 bájtos csomagokból álló, összefüggő adatfolyamok esetén körülbelül 34 Mbps sebességgel képes továbbítani (egyik interfészen fogadni, a másikon küldeni) a forgalmat. A korlát azért jelentkezik, mert az *Ethernet* illesztőprogram által fogadott vagy elküldött csomagok mindegyike egy megszakítást vált ki. A megszakítások kezelése processzoridőt igényel, ami értelemszerűen akadályozza a rendszer egyéb folyamatainak működését.
- Ha a forgalmat 30 Mbps sebességre állítjuk, 18 k / 18 k ***cburst/burst*** érték mellett kapjuk a legjobb átlagos pontosságot.
- A ***löket (burst)*** érték és az elvárt sebességérték között lineáris kapcsolat van. A 30 Mbps sebességhez tartozó ***cburst/burst*** értékek megfelelő kiindulási alapot szolgáltatnak a más sebességekhez tartozó ***burst*** értékek meghatározásához.
- Az átvitel pontosságát a ***cburst/burst*** értékek módosításával lehet szabályozni. A pontos szabályozás sávszélesség-tartománya 64-1500 bájt közötti méretű csomagoknál körülbelül 2 Mbps.
- A sávszélesség egyenlően oszlik meg az adatfolyamok között, amennyiben azok azonos számú bájtot küldenek el, illetve azonos osztályhoz tartoznak.

Linux Journal 2005. március, 131. szám

Yaron Benita az izraeli Jeruzsálemben született, jelenleg San Franciscoban, Kaliforniában él. A Prediwave CMTS szoftverigazgatója, munkája során elsősorban hálózatokkal és beágyazott rendszerekkel foglalkozik. Nős, van egy hat hónapos, gyönyörű kislánya. A yaronb@prediwave.com vagy az ybenita@yahoo.com címen érhető el.

KAPCSOLÓDÓ CÍMEK

- snafu.freedom.org/linux2.2/iproute-notes.html
- tcng.sourceforge.net
- luxik.cdi.cz/~devik/qos/htb/manual/userg.htm
- luxik.cdi.cz/~devik/qos/htb/manual/theory.htm
- www.cisco.com
- www.rns-nis.co.yu/~mps/linux-tc.html

A 2.6-os Linux rendszermag valós idejű és teljesítmény fejlesztései

A Linux rendszermag érzékenységének és valós idejű teljesítményének fejlesztése a jövőben még kedvezőbb fordulatot vehet.

A ***Linux*** rendszermag, minden ***Linux*** terjesztés belső magja, folyamatosan fejlődik. Új módszereket olvaszt magába, növeli a teljesítményét, a skálázhatóságát és használhatóságát. Bár minden rendszermag kiadás újabb alkatrészeket támogat, a nagyobb rendszermag verzióváltások általában komolyabb változásokat jeleznek a rendszermag felépítésében. Ilyen a 2.6-os ***Linux*** rendszermag is. A 2.6-os ***Linux*** rendszermag számos változása jelentősen befolyásolja a ***Linux*** rendszermag teljesítményét az alkalmazott alkatrészektől függetlenül, valamennyi ***Linux*** rendszeren. A 2.6-os rendszermag komoly változásokat hoz a rendszer érzékenységében, jelentősen csökkenti a folyamat és szál-vonatkozású rendszermag többletmunkát valamint a feladat ütemezése és végrehajtása között eltelt időt. Ma már szinte az ipar valamennyi nagyobb ***Linux*** terjesztése a 2003 végén kiadott 2.6 rendszermagon alapul az asztali gépektől a beágyazott rendszerekig. A rendszermag és rendszerteljesítmény rendkívül fontos az olyan koncentrált piacokon mint a beágyazott gépek területe, ahol a magas prioritású folyamatoknak gyakran valós időben kell végrehajtódniuk anélkül, hogy a rendszer félbeszakítaná őket. Ugyanakkor a rendszer teljesítménye és átviteli képessége legalább ilyen fontos a linuxos asztali gépek, és az egyre sikeresebb ***Linux*** vállalati kiszolgálópiacon.

Cikkünkben a valós idejű és rendszerteljesítményt befolyásoló paraméterek természetével foglalkozunk, kiemelve a 2.6 rendszermagba teljesítmény és érzékenység terén bevezetett legfontosabb újításokat. A teljesítmény és az érzékenység továbbra is fontos fejlesztési terület maradt, cikkünkben a Linux teljesítményének és érzékenységének növelését valamint a valós idejű működést célzó néhány aktuális megoldással foglalkozunk. A különféle ***Linux*** rendszermagok és projektek belső, illetve folyamat végrehajtó teljesítményét grafikus teljesítményteszteken jelenítjük meg ahonnan leolvasható, hogyan viselkednek az egyes rendszermagváltozatok különféle terhelési körülmények között.

Lappangás, preemptiv képességek és teljesítmény

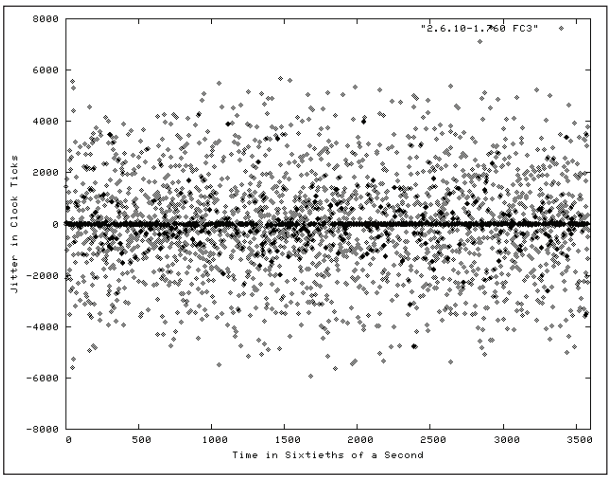
Magasabb teljesítményt gyakorta további és erősebb alkatrészek felhasználásával lehet elérni, azaz komolyabb processzort, nagyobb memóriamennyiséget alkalmazunk. Bár ez egy adatközpontban megfelelő megoldásnak bizonyulhat, jónéhány területen ez nem jelent kielégítő megoldást. A beágyazott ***Linux*** rendszerek például kifejezetten érzékenyek a felhasznált alkatrészek árára. Ezen kívül ha teljesítmény és végrehajtási gondokat memóriavásárlással és más alkatrészekkel próbáljuk megoldani mindössze ideiglenesen elfedjük a problémát. Ahogy a programok igénye növekszik, idővel átlépi a bővített erőforrások határát is, így a probléma ismét a felszínre kerül.

Ezért igen fontos hogy a ***Linux*** rendszerekben alkatrészkerülő módon, a belső rendszer mag javításával érjünk el teljesítménynövekedést. Cikkünk az ilyen típusú ***Linux*** teljesítménymérésekre koncentrál.

A valós idejű rendszerek azon a rendszerek közé tartoznak, ahol a rendszer helyes működése nem csak a kívánt művelet végrehajtásának sebességétől függ, hanem egy bizonyos időzítési kritériumtól is. Kétféle valós idejű rendszer létezik: a lágy és a kemény. A kemény valós idejű rendszerek esetében a kritikus folyamatoknak egy adott időszeletben kell lefutniuk vagy a teljes rendszer hibásnak tekinthető. Klasszikus példa ilyen rendszerre a számítógép vezérlésű autógyújtásvezérlő rendszer – ha a hengerek nem pontosan a jó időben kapnak gyújtást, az autó nem fog működni. A lágy valósidejű rendszerek azok, ahol az időzítési határidőket át lehet lépni a teljes rendszer összeomlása nélkül; a rendszer helyre tud állni az ideiglenes érzékenység csökkenés után.

Mindkét fajta valós idejű rendszer először a magas prioritású feladatokat hajtja végre egy ismert előrejelezhető időszeleten belül. Ez annyit jelent, hogy az operációs rendszer nem végezhet aránytalan többletmunkát a feladat ütemezés, végrehajtás és kezelés során. Ha a folyamatokkal kapcsolatos többletmunka jelentős mértékben növekszik a feladatok számának növekedésével, akkor a teljes rendszerteljesítmény leromlik ahogy egyre több idő szükséges az ütemezés, váltás és újraütemezés elvégzéséhez. A kiszámíthatóság a valós idejű operációs rendszerek egyik legfontosabb jellemzője. Ha nem tudjuk kiszámítani a rendszer teljesítményigényét egy tetszőleges időpillanatban, nem tudjuk biztosítani, hogy az adott folyamat szükség esetén előre látható lappangással indul el vagy indul újra, illetve hogy az előírt időszeletben befejeződik-e.

A 2.6-os ***Linux*** rendszermagban új típusú ütemező jelent meg, amelynek végrehajtási ideje nem függ az ütemezett



1. ábra A dobozos Fedora Core rendszermag remegési értékei

feladatok számától. Ez a megoldás ***O(1)*** ütemezőként ismert a nagy-***O*** algorimus jelölési rendszerben, ahol az ***O*** az angol ***Order*** szó rövidítése, a zárójeles szám pedig a legrosszabb teljesítmény felső határa az algoritmusban résztvevő elemek függvényében. Az ***O(N)*** tehát annyit tesz, hogy az algoritmus hatékonysága az elemek számától függ, az ***O(1)*** jelentése pedig, hogy az algoritmus, azaz jelen esetben az ütemező futásigénye minden esetben azonos, azaz független a résztvevő elemek számától.

Az operációs rendszernek kiadott feladatvégrehajtási utasítás és a feladat tényleges elindulása között eltelt időt lappangásnak nevezzük. A folyamatvégrehajtás nyilvánvaló módon függ az adott feladat fontossági szintjétől, de azonos fontosságot feltételezve, azt az időt amelyet az operációs rendszer a feladat ütemezésével és futtatás megindításával tölt, a rendszerütemező többletmunkája és a rendszer által végzett egyéb tevékenységek határozzák meg. Amikor feladatot ütemezünk a rendszer futtatási sorába helyezve, a rendszer megvizsgálja, hogy a folyamat magasabb prioritású-e mint az éppen futó folyamat prioritási szintje.

Ha igen, a rendszermag megszakítja az éppen futó folyamatot és átvált az új folyamat környezetére. A rendszermagban az éppen futó folyamat megszakítását és a környezetváltást nevezzük ***preempciónak***.

Sajnos, a rendszermag nem mindig használhatja a preemptivitást. Az operációs rendszer magja gyakran igényel kizárólagos erőforrás és belső adatszerkezet elérést, hogy fenntarthassa saját belső szerkezetének sérülésmentességét. A ***Linux*** rendszermag régebbi verzióiban az erőforrásokhoz való kizárólagos hozzáférést gyakran forgózárakkal biztosították. Ez ennyit jelentett, hogy a rendszermag egy rövid ciklusban lépkedett, amíg az adott erőforrás elérhetővé nem vált, vagy amíg használatban volt, ezzel persze megnövelve minden más feladat lappangását amíg a rendszermag be nem fejezte a feladatát.

A rendszermag preemptivitásának felbontása folyamatosan fejlődött az elmúlt nagyobb rendszermag verziókban. Például a beágyazott ***Linux*** és eszköz gyártótó ***TimeSys***-től származó ***GPL 2.4 Linux*** rendszermag, egy korábbi alacsony lappangású ütemezőt és teljesen preemptív rendszermagot tartalmazott. A 2.4-es ***Linux*** rendszermag sorozatban, ***Novell/Ximian***-tól ***Robert Love*** kiadott egy jól ismert rendszermag foltot amely magasabb preemptivitást biztosított és amelyet a hagyományos rendszermag forrásra is alkalmazni lehetett. Más foltok, mint például az 1995 óta belső rendszermagíró ***Molnár Ingo*** alacsony lappangási foltja, tovább fokozták e folt teljesítményét és csökkentették a rendszermag lappangási idejét.

A ***TimeSys*** termékének és az említett foltok kulcselképzelése az volt, hogy a ***forgózárakat mutexekkel (mutual exclusion mechanism, azaz kölcsönös eléréskizárás)*** kell helyettesíteni ahol csak lehet. Ezek ugyanis anélkül biztosították a rendszermag erőforrás és integritás biztonságát, hogy a rendszermagot leállították és várakozásra kényszerítették volna. Ezen úttörő foltok által használt megoldások ma már a 2.6 ***Linux*** rendszermag szerves részét képezik.

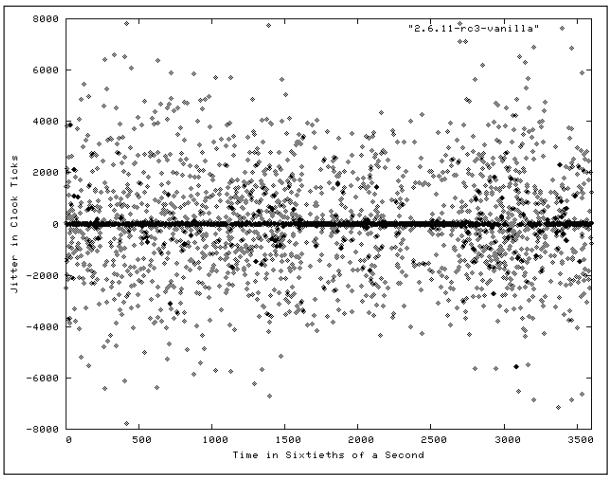
Valós idejű megoldások Linux alatt

Három valós idejű linuxos megoldás létezik jelenleg: az ***RTAI Projekt*** által alkalmazott kettős rendszermag modell, amelyet beágyazott ***Linux*** gyártók, például az ***FSMLabs*** termékeiben alkalmaznak; egy beágyazott ***Linux*** gyártó, a ***MontaVista*** szárnyai alatt fejlődő ***Real-time Linux Projekt***; valamint a szabadon elérhető és használható preemptivitási és valós idejű munka, amelyet ***Molnár Ingó*** és más fejlesztők neve fémjelez, s amelyet nyíltan vitatnak meg a ***Linux*** rendszermag levelezőlistán, és amelyre a ***MontaVista Projekt*** is épít. Ezeken az alap rendszermag modelleken felül más kiegészítő projektek, például a robosztus mutexek és a nagy felbontású időzítők is jelentős mértékben hozzájárultak a valósidejű alkalmazások teljes körű támogatásának fejlődéséhez ***Linux*** alatt.

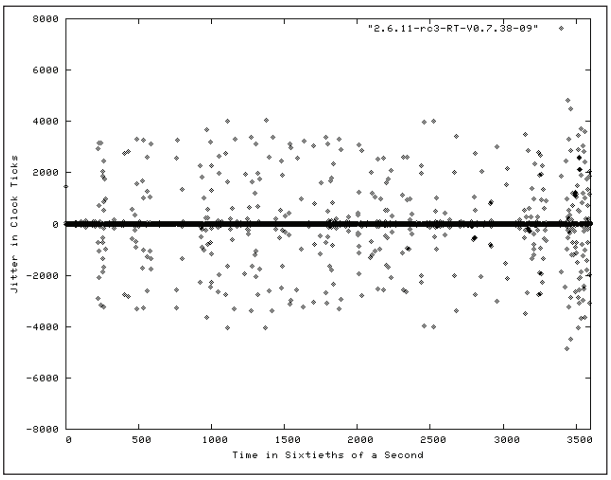
A valós idejű kettős rendszermag elgondolás igen érdekes megközelítése a Linux alatti valós idejű alkalmazáskezelés problémakörének. Ebben az elképzelésben a rendszer tulajdonképpen egy apró, nem ***Linux*** valósidejű rendszermagot futtat, amely viszont a ***Linuxot*** futtatja a lehető legalacsonyabb prioritási szinten. A valósidejű alkalmazások, amelyeket kifejezetten ehhez a nem-linuxos rendszermaghoz írtak a hozzá tartozó valósidejű csatolófelület felhasználásával, ebben a rendszermagban hajtódnak végre, de magasabb szinten mint a ***Linux*** vagy bármely ***Linux*** alkalmazás. Ugyanakkor képesek adatokat cserélni a ***Linux*** alkalmazásokkal. Igaz, műszakilag nagyon érdekes elgondolás, amellyel valósidejű alkalmazásokat futtathatunk ***Linux*** rendszer mellett, azonban teljességben kikerüli az általános ***Linux*** rendszermag preemptivitás és teljesítmény fokozási problémákat. Éppen ezért a ***Linux*** mag fejlődésének szempontjából nem is olyan érdekes.

A ***MontaVista Projektre***, azaz a másik valós idejű ***Linux*** megoldásra igen nagy hatással volt Molnár Ingo és más rendszermagfejlesztők által végzett már meglévő munka, azonban tartalmaz néhány egyéb prototípus foltot, amelyek kizárólag a ***MontaVista*** weblapján érhetőek el. A jelenlegi foltok itt a 2.6.9-es ***Linux*** rendszermaghoz valók (***rc4***). Ezért aztán nem igazán alkalmazhatók probléma mentesen a hivatalos ***Linux*** rendszermag kiadásokra, amelyek írásunk születésekor már a 2.6.1-es verzió felé járnak. Így aztán e projektből származó eredményeket nem tudtuk megmutatni ebben a cikkben.

Molnár Ingó az ***O(1) Linux*** ütemező atyja és más fejlesztők munkája az ütemezőn és a preemptivitási képességeken igen komoly lendülettel haladnak. Miközben kibővítik



2. ábra Remegési eredmények egyszerű 2.6.11-rc3 Rendszermaggal



3. ábra Remegési eredmények valós-idejű/Preemptív folttal ellátott 2.6.11-rc3 Rendszermag esetében

a ***Linux*** rendszermag lehetőségeit, naprakész foltokat nyújtanak, amelyek a hatékonyabbá varázsolják a rendszerütemezést, csökkentik a lappangási időt és tovább növelik a preemptivitást.
Ezeket a foltokat nagy figyelemmel kísérik számtalan csoport és szervezet fejlesztői, beleértve a ***Raytheont***, olyan beágyazott ***Linux*** fejlesztőket mint a ***TimeSys*** valamint a ***Linux audió*** közösség. A foltok megnövelik a rendszerérzékenységet és minimálisra csökkentik a megszakítások hatását, két részre osztva a megszakításkezelőket: egy közvetlen alkatrész válaszra és egy ütemezhető megszakítás feldolgozó szakaszra. Mint a nevük is mutatja a megszakítások olyan kérelmek, amelyek azonnali figyelmet igényelnek a rendszertől. Az ütemezhető megszakítás kezelési megoldás, vagy ismertebb nevén soft IRQ, a lehető legkisebbre csökkenti az általános rendszer reakcióidő és teljesítmény.
A következő rész ábrái különféle sima rendszermagok teljesítményét érik össze Molnár Ingó és mások által készített ütemezési, preemptivitási foltokkal javított változatok képességeivel. Ezek a foltok naprakészek és teljes ***Linux*** rendszermag fejlesztéseket tartalmaznak melyek közvetlen előnyöket nyújtanak bármely ***Linux*** felhasználónak aki projektjébe szeretné építeni őket.

Teljesítménymérés

2002-ben, a ***Linux Journal*** weblapon megjelentet egy cikk ***Andrew Webber*** tollából ***„Realfeel Test of the Preemptible Kernel Patch”*** címmel. Ez a cikk ***Mark Hahn Realfeel*** nevű nyílt teljesítménymérő programját használta „hogy összehasonlítsa a preemption és érzékenység értékeket az eredeti ***Linux*** 2.4-es és ***Robert Love*** preemptivitási foltjával javított rendszermagot. A Realfeel periodikusan megszakításokat hoz létre, majd a számítógép mért válaszidejét összehasonlítja a rendszer optimális válasz idejével. A megszakításkérés kiadása és annak kezelése közt eltelt idő a lappangás, az előrejelzett és a tényleges lappangás között eltelt idő a remegés (jitter). Ezt a remegés értéket gyakran használják a rendszer válaszidejének mérésére és összehasonlítására. Cikkünk ugyanazt a teljesítménymérő programot használja mint amelyet ***Webber*** cikkében megismerhettünk, ám mérés közben lényegesen nagyobb terhelést tettünk a gépre. Valós idejű operációs rendszerek tesztelése során gyakran alkalmazzák ezt a módszert, hiszen még a nem valós idejű rendszerek is alacsony lappangást mutathatnak amikor terhelés nélkül vagy alacsony terhelés mellett működnek. A következő fejezet rajzai az eredményeket is kicsit másképp ábrázolják, hogy könnyebb legyen elképzelni és összehasonlítani a különféle ***Linux*** rendszerek lappangási idejének különbségeit.

Mérési eredmények

A fejezeteben található eredmények egy közepes teljesítményű ***Pentium*** osztályú rendszeren készültek, amelyet egyetlen 1.7GHz ***AMD Athlon*** processzor hajtott meg, 512MB rendszermemória mellett. A rendszer ***GNOME*** asztalkezelő környezetet és ***Fedora Core 3 Linux*** terjesztésben szokásos rendszerfolyamatokat futtatta, 2004 február 10.-ei legfrissebb foltokkal. A kipróbált rendszerek: eredeti 2.6.10 ***Linux*** rendszermag, a ***2.6.10-1.760_FC3*** rendszermag, amelyet a ***Fedora Core 3*** frissítéssel érhetünk el, eredeti ***2.6.11-rc3*** rendszermag és a ***2.6.11-rc3*** rendszermag ***Molnár Ingo*** legújabb valós idejű preemptivitási foltjával. Minden rendszermagot azonos rendszermag beállításállomány mellett fordítottuk, eltekintve persze az új rendszermagforrásban felbukkanó újabb beállításoktól.
Az olyan több folyamatos rendszerekben mint a ***Linux***, a rendszer soha nem alszik. Bizonyos rendszerfolyamatok, például az ütemező, állandóan futnak. Amennyiben grafikus felületet használunk (GUI) és kezelőfelületeket mint a ***KDE***, a ***GNOME*** vagy akár az egyszerű ***X Window*** rendszer, az ablakkezelők folyamatosan várják a beérkező eseményeket. Mivel az igazi preemptivitást és valós idejű teljesítményre voltunk kíváncsiak, pár alkalmazás elindításával megterheltük a rendszert, miközben az egyes teljesítménymérő tesztek eredményeit gyűjtöttük. Mint korábban említettük, a rendszer ***GNOME***-ot futtatott négy nyitott ***xterm*** mellett – ezek közül egyben futott a ***realfeel*** teljesítménymérő a másikban pedig egy olyan folyamat futott, amely állandóan ***ls*** és ***find*** parancsokat hívogatott a rendszer gyökérpartícióján, végül a maradék kettőben külön forráskönyvtárban található ***Linux 2.6.x*** rendszermagok fordultak, ***clean*** állapotból.

Az *1. ábra* a ***Realfeel*** teljesítménymérő grafikonján ábrázolja egy dobozos ***Fedora Core*** rendszeren futtatva egy percen keresztül. A rendszer ***2.6.10-1.760_FC3*** rendszermagot futtatott, amely lényegében egy 2.6.10-es rendszermag néhány ***Red Hat*** folttal és fejlesztéssel. A pontok a megszakítás kérés és kezelés közötti remegést (jitter) ábrázolják. Az X tengely a mintaidő 1/60-ad másodpercben. A negatív remegési értékek olyankor fordulnak elő amikor a rendszer gyorsabban válaszolt mint az előrejelzett válaszidő. Amint azt az ábrán láthatjuk, a megszakítási kérelmek legtöbbje a várakozásoknak megfelelő idő alatt készült fel, így eredményképpen egy jól kivehető sörtét vonalat kapunk az Y tengely 0 értéke körül.
A *2. ábra* ugyanezen rendszer ***Realfeel*** programmal mért eredményeit mutatja sima ***2.6.11rc3*** rendszermaggal, amely a következő 2.6.11 rendszermag 3. kiadási jelöltje. Azt eredményeket ismét csak egy perc alatt gyűjtöttük. Amint azt az eredményekből láthatjuk, a ***2.6.11-rc3*** rendszermag jobb teljesítményt nyújtott mint az ***FC3***-as rendszermag, sokkal több helyen tapasztalhatjuk, hogy a megszakítás kérelem és kezelés közötti remegés 0 értékű.
A *3. ábrán* ***Molnár Ingó*** valós-idejű/preemtivitás foltjaival bővített ***2.6.11rc3*** rendszermagot használó, de egyébként változatlan rendszer eredményeit látjuk ***Realfeel*** programmal mérve. Ezeket az eredményeket is egy perc alatt gyűjtöttük össze, az előzőekkel azonos terhelésgenerátor mellett. Amint azt az eredményekből láthatjuk, a valós-idejű/preemptivitás folt esetében lenyűgözően jó remegési eredményeket kaptunk, és viszonylag kevés helyen láthatunk eltéréseket a becsült időintervallum értékekhez képest. Rendszerünkön mindez nagyobb érzékenységet eredményez, a programfutás jóval kiszámíthatóbb mint azt egy sima ***FC3*** vagy eredeti ***2.6.11-rc3*** rendszermag esetében várnánk.

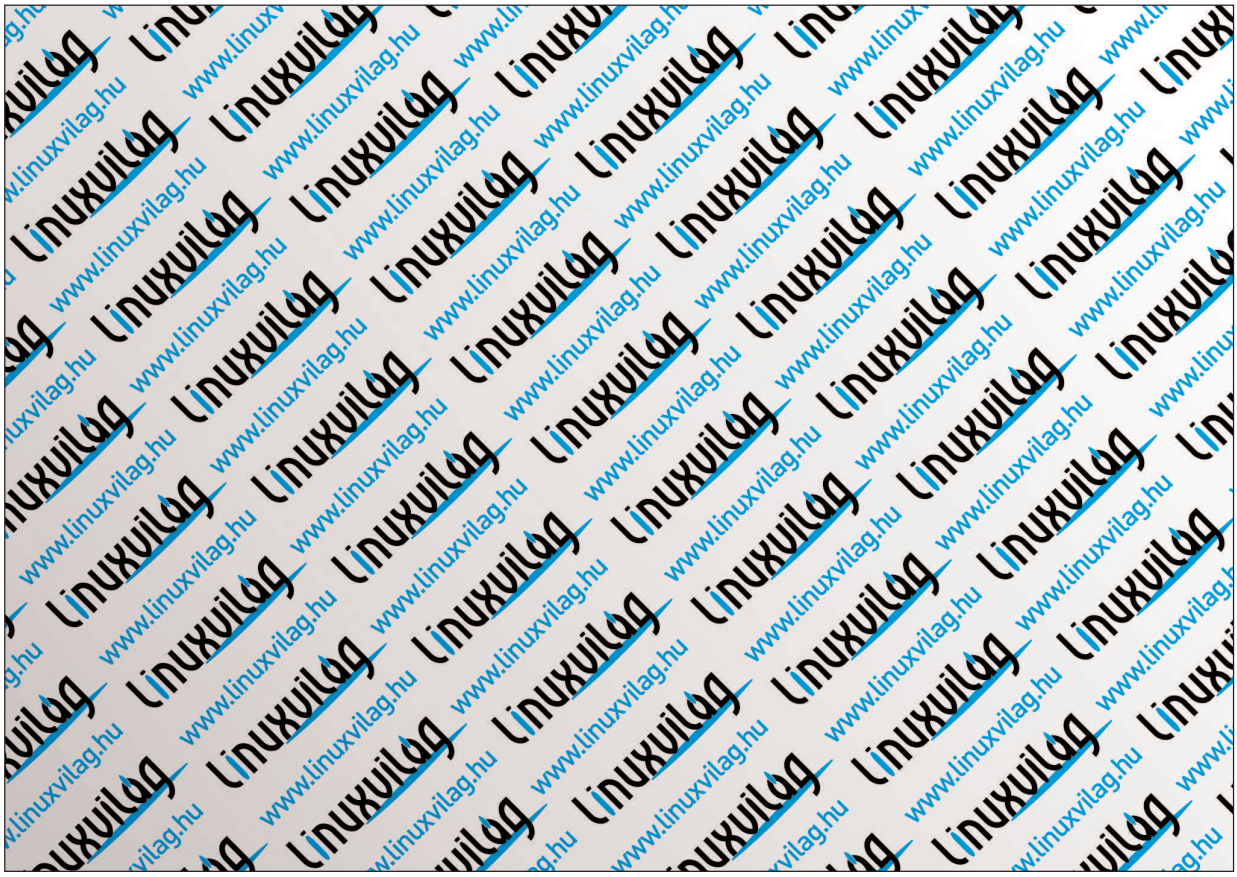
Összefoglalás

A 2.6-os ***Linux*** rendszermag javított ütemezése, ***SMP*** és skálázási fejlesztései nagyobb teljesítményű ***Linux*** rendszereket eredményeztek mint bármikor azelőtt. Jobban ki tudják használni a rendszererőforrásokat valamint kiszámíthatóbban futtatják a rendszermagot és a felhasználói folyamatokat a rendszer igényei szerint. Léteznek további javítások is, ám ezeket jelenleg csak a rendszermag kézi foltozásával használhatjuk, vagy olyan gyártótól kell ***Linux*** terjesztésünket beszerezni, mint a ***TimeSys***, amely már beépítette és teszteli ezeket a nagyteljesítményű foltokat.
Kész csoda, hogy egyáltalán létezik ilyen szabad, nyílt forrású rendszermag és robosztus végrehajtási környezet mint a ***GNU/Linux***. Most, hogy egyéni fejlesztők mellett már cégek is fejlesztik teljesítményét, még fényesebb jövőt jósolhatunk neki. Ezek, és a ***Linux*** egyéb módosításai felvetik és érvéként szolgálhatnak a ***Linux*** mint javasolt operációs rendszer mellett a beágyazott rendszerek, kiszolgálók és asztali gépek területén.

Linux Journal 2005. június, 134. szám

A cikk forrása: ➲ ***www.linuxjournal.com/article/8199***

William von Hagen



Dinamikus megszakításkérés-kiosztás illesztőprogramokhoz

A megszakítás az az eszköz, amellyel a hardver felhívja magára a szoftver figyelmét. Vizsgáljuk meg a működését!

Egy számítógép a vele szemben támasztott elvárásoknak csak akkor tud megfelelni, ha külső eszközökkel is képes kapcsolatba lépni. Az eszközök és a processzor közötti kapcsolattartáshoz az átjárót a megszakítások biztosítják. A megszakításkérési vonalak eszközökhöz való hozzárendelése és a megszakítások kezelésének módja kulcsszerepet játszik az illesztőprogramok fejlesztésében. Mivel a megszakításkérési vonalak száma korlátozott, nagyobb számú eszközhöz csak a megszakítások megosztásával biztosíthatunk hozzáférést. Ugyanakkor, ha megpróbálunk kiosztani egy már használatban lévő megszakítást, a rendszer összeomlik. Írásomban a megszakításokkal és a megszakításkezeléssel kapcsolatos alapismereteket szeretném átadni, valamint be szeretnék mutatni egy karakteres eszközökhöz használható ***megszakításkérés-kiosztó (IRQ)*** megoldást.

Minden eszköz célja valamilyen hasznos feladat ellátása, és ennek teljesítéséhez kapcsolatot kell tartania a processzorral. Ha a processzor közölni szeretne valamit az egyik eszközzel, akkor a megfelelő eszközvezérlőnek küldi el utasításait. Az eszközvezérlő irányítja az eszköz működését. Hasonlóan, ha az eszköz válaszolni szeretne a processzornak, mert például a kért adatok készen állnak a beolvasásra, akkor egy megszakítás létrehozásával vonja magára a processzor figyelmét. A megszakítás egy hardveres, a processzor és az eszköz közötti kommunikációt lehetővé tévő megoldás.

A ***Linux*** egészen a 2.6-os változatig nem preemptív működésű volt, ami azt jelenti, hogy ha egy folyamat rendszermag módban fut, és a futtatásra kész folyamatok várólistájára felkerül egy nagyobb fontosságú folyamat, akkor a kisebb fontosságú folyamattól nem lehet elvenni a futás jogát, amíg vissza nem tér felhasználó módba. A megszakítások azonban akkor is elvonhatják a processzor figyelmét, amikor éppen egy rendszermag módban futó folyamattal foglalkozik. Ezzel a megoldással növelni lehet a rendszer átviteli sebességét. Amikor megszakítás érkezik, a processzor felfüggeszti az éppen futó folyamatot, majd a megszakítást okozó esemény jellegétől függően lefuttatja a megfelelő kódot.

A számítógép minden eszköze rendelkezik egy eszközvezérlővel, valamint egy olyan hardveres érintkezővel, amelyet akkor használ jelzésre, amikor szüksége van a processzor közreműködésére. Ez az érintkező a processzor megfelelő megszakításlábához csatlakozik, ezen keresztül folyik az adatcsere. A processzor vezérlőhöz csatlakozó lábát megszakításkérési vonalnak nevezzük. Egy processzor több ilyen lábbal is rendelkezik, vagyis több eszközzel is képes tartani a kapcsolatot. A korszerű operációs rendszereknél egy ***programozható megszakításvezérlő (programmable interrupt controller, PIC)*** kezeli a processzor és az eszközvezérlők közötti ***IRQ***-vonalakat. Az egyes rendszerekben a szabad ***IRQ***-k száma korlátozott, ám a ***Linux*** rendelkezik egy olyan megoldással, amelynek révén több eszköz is osztozhat ugyanazokon a megszakításokon.

A megszakítások használata a programozók munkájához hasonlítható. A programozó megnyitja a postaládáját, majd nekikezd szokásos munkájának. Amikor új levele érkezik, akkor rövid hangjelzést hall, valamint a képernyő alján valamilyen egyéb jelzés is megjelenik. Azonnal elmenti a programot, amin éppen dolgozik, majd átvált a postaládára. Elolvassa a levelet, küld egy visszaigazolást, majd folytatja imént abbahagyott munkáját. Az általa végrehajtott lépések részletes listáját valamikor később küldi el. Hasonlóan, amikor egy processzor futtat egy folyamatot, az eszközök az egyes feladatokhoz kapcsolódóan megszakításokat küldhetnek neki, például jelezve, hogy valamilyen adat készen áll az átvitelre. Amikor befut egy megszakítás, a processzor azonnal elmenti a programszámláló aktuális értékét a rendszermag módú verembe, majd végrehajtja a megfelelő ***megszakításkiszolgáló eljárást (interrupt service routine, ISR)***. Az ***ISR*** egy függvény a rendszermagban, feladata a megszakítás jellegének meghatározása és a szükséges művelet végrehajtása, például egy adatblokk másolása a merevlemezről a központi memóriába. Az ***ISR*** lefuttatása után a processzor visszaállítja a korábbi folyamatot, és folytatja a futtatását.

Az illesztőprogram egy a rendszermagba beépülő, az alkalmazások kéréseit fogadó programmodul. Amikor egy alkalmazás adatokat szeretne olvasni egy eszközről, akkor azonnal meghívásra kerül a megfelelő illesztőprogram, majd megnyílik olvasásra a kívánt eszköz. Ha a rendszer sokat vár a lassú eszközökre, akkor nem tud a feladataival foglalkozni. A rendszermag fejlesztőinek egyik fő célja az erőforrások minél jobb kihasználásának biztosítása. Mivel nem akar a hardvereszközök által szállított adatokra várakozni,

a rendszermag kiadja a feladatot az eszközvezérlőnek, majd visszatér a felfüggesztett folyamathoz. Amikor az olvasás befejeződött, az eszköz egy megszakítás segítségével értesíti a processzort, amely végrehajtja a megfelelő *ISR*-t.

A megszakítások osztályozása

A megszakításokat két fő kategóriára osztjuk, szinkron és aszinkron megszakításokra. A szinkron megszakításokat a processzor vezérlőegysége hozza létre valamilyen utasítás végrehajtásakor. A vezérlőegység akkor adja ki a megszakítást, amikor végzett az utasítások végrehajtásával, innen származik a szinkron jelző. Az aszinkron megszakításokat a hardvereszközök hozzák létre, véletlenszerűen, bár a processzoróra állásához igazodva. Az *Intel* alapú gépeknél az első típust kivételnek, a másodikat pedig megszakításnak nevezzük. A megszakításokat egy előjel nélküli, egybájtos egész azonosítja, ezt vektornak nevezzük. A vektortartomány a 0 - 255. Az első 32 (0 - 31 sorszámú) vektor a kivételeket, más néven nem maszkolható megszakításokat azonosítja, ezekről *„Linuxos jelzések alkalmazásfejlesztői szemszögből”* (*Linuxvilág*, 2003. július) című cikkemben volt szó. A 32 - 47 tartomány a maszkolható megszakításoké, ezeket hozzák létre az *IRQ*-k (a 0 - 15 számú *IRQ*-vonalak).
Az utolsó tartomány a 48 - 255, ebbe a szoftveres megszakítások tartoznak. Ilyen például a rendszerhívások megvalósítására használt 128-as megszakítás (*int 0X80 assembly* utasítások).

IRQ-kiosztás

A rendszer már használatban lévő megszakításairól a */proc* könyvtárban találunk pillanatképet. A `$cat /proc/interrupt` parancs a megszakításokkal kapcsolatos adatokat jeleníti meg. A saját gépemen a következő volt a kimenete:

```
CPU0
  0:   82821789          XT-PIC  timer
  1:        122          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  8:          1          XT-PIC  rtc
 10:     154190          XT-PIC  eth0
 12:        100          XT-PIC  i8042
 14:      21578          XT-PIC  ide0
 15:         18          XT-PIC  ide1
NMI:          0
ERR:          0
```

Az első oszlopban az *IRQ*-vonalak száma (32 - 47 vektortartomány) látható, a másodikban pedig az, hogy az egyes megszakítások a rendszer elindítása óta hányszor jutottak el a processzorhoz. A harmadik oszlop a *PIC*-kel kapcsolatos, az utolsó pedig azoknak az eszközöknek a nevét tartalmazza, amelyek kezelőt jegyeztek be az egyes megszakításokhoz.
Ha egy illesztőprogramot dinamikusan akarunk betölteni, akkor először használaton kívüli *IRQ*-vonalat kell találnunk a rendszerben. A `request_irq` függvény segítségével meghatározott azonosítójú *IRQ*-vonalat rendelhetünk az eszközünkhöz. A `request_irq` szintaxisának meghatározása a *linux/sched.h* fájlban található:

```
int
request_irq (unsigned int irq,
             void (*handler) (int, void *,
                              struct pt_regs *),
             unsigned long flags,
             const char *device, void *dev_id);
```

A függvény átadott értékeinek szerepe a következő:

- `unsigned int irq`: A rendszertől kért megszakítás száma.
- `void (*handler) (int, void *, struct pt_regs *)`: Megszakítás kérésekor gondoskodnunk kell a kezelését végző *ISR*-ről, egyébként a processzor egyszerűen nyugtázza, majd nem történik semmi. Az átadott érték a kezelőfüggvényre mutat. A kezelőfüggvény szintaxisa a következő:

  ```
  void
  handler (int irq, void *dev_id,
           struct pt_regs *regs);
  ```

  Az első átadott érték az *IRQ* száma, erről a `request_irq` függvény kapcsán már volt szó. A második átadott érték egy fő- és egy alrészből álló eszközazonosító, ez adja meg az aktuális megszakítási esemény kezelését végző eszközt. A harmadik átadott értéket a folyamat környezetének a rendszermag verembe mentésére használjuk, mielőtt a processzor megkezdené a megszakításkezelő függvény futtatását. A korábbi folyamat visszaállításakor a rendszer ezt az adatszerkezetet használja fel. Normál esetben az illesztőprogramok íróinak ezzel az átadott értékkel nem kell foglalkozniuk.
- `unsigned long flags`: A `flags` változó a megszakításkezelésben jut szerephez. Az `SA_INTERRUPT` jelzőt gyors megszakításkezelőknél használjuk, ez minden maszkolható megszakítást letilt. Az `SA_SHIRQ`-t akkor használjuk, ha az *IRQ*-t egynél több eszköz közt akarjuk megosztani; az `SA_PROBE` jelzőt akkor, ha az *IRQ*-vonal segítségével le szeretnénk kérdezni egy hardvereszközt; az `SA_RANDOM` jelző pedig a rendszermag véletlenszám-generátorának kezdeti értékadására használható. A jelzőről további részleteket a */usr/src/linux/drivers/char/random.c* fájlban találni.
- `constant char *device`: Az IRQ-t használó eszköz neve.
- `void *dev_id`: Az eszköz azonosítója, egy mutató az eszköz adatszerkezetére. Ha a megszakítás meg van osztva, akkor ez a mező meghatározott eszközre mutat.

A `request_irq` függvény siker esetén nullával, a foglalás sikertelenségekor pedig `-EBUSY` értékkel tér vissza. Az `EBUSY` a 16-os számú hiba, leírása a */usr/src/linux/include/asm/errno.h* fájlban található. A `free_irq` függvény elengedi az adott eszköz *IRQ*-ját. Szintaxisa a következő:

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

Az argumentumok jelentése a fentiekkel egyezik meg.
ISR meghívására megszakítás fellépésekor kerül sor. A megszakítás hatására végrehajtandó műveletek leírása az *ISR*-ben szerepel. A rendszermag fenntart a memóriában egy a megszakítási eljárások címeit (megszakításvektorokat)

1. kódrészlet my_module.c

```
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/interrupt.h>
static struct file_operations fops;
static int Major, irq = 7;
static void OurISR (int irq, void *device,
                    struct pt_regs *regs)
{
  /* fontos és azonnali, időkritikus
  ➥feladatok */
}
static int __init my_init_module(void)
{
    int status;
    Major = register_chrdev(0, “OurDevice”,
    ➥&fops);
    if (Major == -1) {
        printk (“ A dinamikus főazonosító “
                “foglalása sikertelen.\n”);
        return Major;
    }
    status = request_irq(irq,
                         (void *)OurISR,
                         SA_INTERRUPT,
                         “OurDevice”, &fops);
    if (status == -EBUSY) {
        printk (“Az IRQ-azonosító foglalása
        ➥sikertelen.\n”);
        unregister_chrdev(Major, “OurDevice”);
        return status;
    }

    printk (“A modul sikeresen betöltve.\n”);
    printk (“Az eszköz főazonosítója:   %d\n”,
            Major);
    printk (“Az eszköz IRQ-ja:     %d\n”,
            irq);
    return 0;
}
static void __exit my_cleanup_module (void)
{
    printk(“A(z) %d főazonosító és a(z) %d
    ➥IRQ-azonosító “
            “elengedése megtörtént.\n”, Major,
            ➥irq);
    free_irq(irq, &fops);
    unregister_chrdev(Major, “OurDevice”);
    printk(“A modul sikeresen eltávolítva.\n”);
}
module_init (my_init_module);
module_exit (my_cleanup_module);
MODULE_LICENSE(“GPL”);
```

tartalmazó táblázatot. Amikor egy megszakítás érkezik, a processzor lekérdezi az ***ISR*** címét a megszakításvektorok táblázatából, majd lefuttatja. Az ***ISR*** feladata, hogy a megszakítás jellegétől – mint például írás vagy olvasás – függően válaszoljon az eszköznek. Általában az ***ISR*** alvó folyamatokat ébreszt fel az eszközön, ha a megszakítás olyan eseményt jelez, amire éppen várnak.

Azt az időt, amit a processzor egy-egy megszakítás kezeléséhez igényel, megszakítási lappangási időnek nevezzük. A megszakítási lappangási idő része a hardveres jelterjedési idő, a regiszterek elmentésének ideje és a szoftveres terjedési idő. A rendszer teljesítményének javításához a megszakítási lappangási időt a lehető legkisebbre kell csökkenteni, ezért az ***ISR***-nek rövidnek kell lennie, és a megszakításokat csak rövid időre tilthatja le. A megszakítások letiltása alatt is jelentkezhetnek további megszakítások, ám a processzor figyelmen kívül hagyja őket, amíg a megszakítások újraengedélyezése meg nem történik. Ha egynél több megszakítás van blokkolva, a processzor fontossági sorrendben engedélyezi őket, amikor késszé válik a kezelésükre.

Az illesztőprogramok készítőinek az illesztőprogramok kódjában csak akkor szabad letiltaniuk a megszakításokat, ha valóban szükséges, mert ilyenkor a rendszer nem frissíti az időzítőket, nem továbbítja a hálózati csomagokat a pufferek felé és felől, és így tovább. Az ***ISR***-eket úgy kell megírni, hogy más folyamatok számára is biztosítsák a processzor elérhetőségét. A való világban viszont az ***ISR***-ek hosszas feladatokat kezelnek. Ilyenkor az ***ISR*** csak az időkritikus adatcseréket végezheti el a hardverrel, és a ***taskletet*** kell használnia a tényleges adatátvitel lebonyolítására. A ***tasklet*** a legújabb ***Linux*** rendszermag egy különleges szolgáltatása, amely a megszakításokkal kapcsolatos műveletek egy részének megfelelő időben való elvégzésére alkalmas. A ***tasklet*** a szoftveres megszakítás, és más megszakítások által megszakítható. A megszakítások belső világáról ***Bovet*** és ***Cesati*** írásából (lásd az internetes forrásokat) lehet tájékozódni, a megszakítások megvalósítását – az illesztőprogramok nézőpontjából szemlélve – pedig ***Rubini*** és ***Corbet*** tárgyalja.

Egy egyszerű megvalósítás

A rendszermagmodulok illesztőprogramokat tartalmaznak, ezeket a meglévő rendszermaghoz, akár a rendszer működése közben lehet betölteni. A dinamikus ***IRQ***-kiosztást az ***1. kódrészletben*** szereplő egyszerű modullal szeretném szemléltetni. Egy egyszerű karakteres eszközhöz készült illesztőprogramról van szó, az eszközt nevezzük ***OurDevice***-nak (***SajátEszköz***), ehhez foglalunk dinamikusan ***IRQ***-vonalat. A modul beillesztésekor az init_module függvény fut le. Ha a foglalás sikeres, akkor kiírjuk az eszköz főazonosítóját és a kapott ***IRQ*** számát. Ettől a pillanattól az ***IRQ*** kiosztását a ***/proc*** könyvtárban is ellenőrizhetjük. A kapott ***IRQ***-t a modul eltávolításakor elengedjük. ***IRQ***-azonosítót a kódon belül legjobban egy nyílt belépési ponton lehet bejegyezni, amely a megfelelő pillanatban az elengedésről is gondoskodik.

A ***my_module.c*** fájlt a 2.6.0-0.test2.1.29 rendszermag alatt fordítottam le. A ***kernel-2.6.0-0.test2.1.30.i586.rpm*** fájlt az összes szükséges ***RPM***-el együtt kell letölteni és telepíteni. Az ***RPM***-et a ***people.redhat.com/arjanv/2.5/RPMS.kernel*** címen érhetjük el, az illesztőprogram fordítását pedig a következőképpen kell elvégeznünk:

```
gcc -Wall -O3 -finline-functions \
-Wstrict-prototypes -falign-functions=4 \
-I/lib/modules/2.6.0-0.test2.1.29/build/include \
-I/lib/modules/2.6.0-0.test2.1.29/build/include/
?asm/mach-default
-I./include -D__KERNEL__ -DMODULE -DEXPORT_SYMTAB \
-DKBUILD_MODNAME=my_module -c my_module.c -o \
my_module.o
```

A *my_module.o* betöltése után – ha a főazonosító és az *IRQ* foglalása sikeres – a megfelelő üzenetek jelennek meg. Ha az *IRQ*-azonosítót egy másik eszköz már használta, akkor a rendszermag törli az eszköz bejegyzését, és felszabadítja a főazonosítót. A `$cat /proc/interrupt` parancs a következő kimenetet jeleníti meg:

```
           CPU0
  0:   82887219          XT-PIC  timer
  1:        122          XT-PIC  i8042
  2:          0          XT-PIC  cascade
  7:          0          XT-PIC  OurDevice
  8:          1          XT-PIC  rtc
 10:     154769          XT-PIC  eth0
 12:        100          XT-PIC  i8042
 14:      21636          XT-PIC  ide0
 15:         18          XT-PIC  ide1
NMI:          0
ERR:          0
```

A kimenetben látható az `OurDevice`-hoz tartozó bejegyzés, illetve az eszköz *IRQ*-ja. Amikor a modult eltávolítjuk, a rendszermag felszabadítja az *IRQ*-azonosítót és a főazonosítót, valamint törli a készülék bejegyzését.

Összefoglalás

Remélem, hogy sikerült tisztáznom a megszakításokkal és a kezelésükkel kapcsolatos alapokat. A `request_irq` és a `free_irq` függvényt elsősorban illesztőprogramok készítésekor érdemes ismerni, a dinamikus *IRQ*-foglalási eljárást pedig egy egyszerű karakteres eszköz illesztőprogramjával szemléltettük.

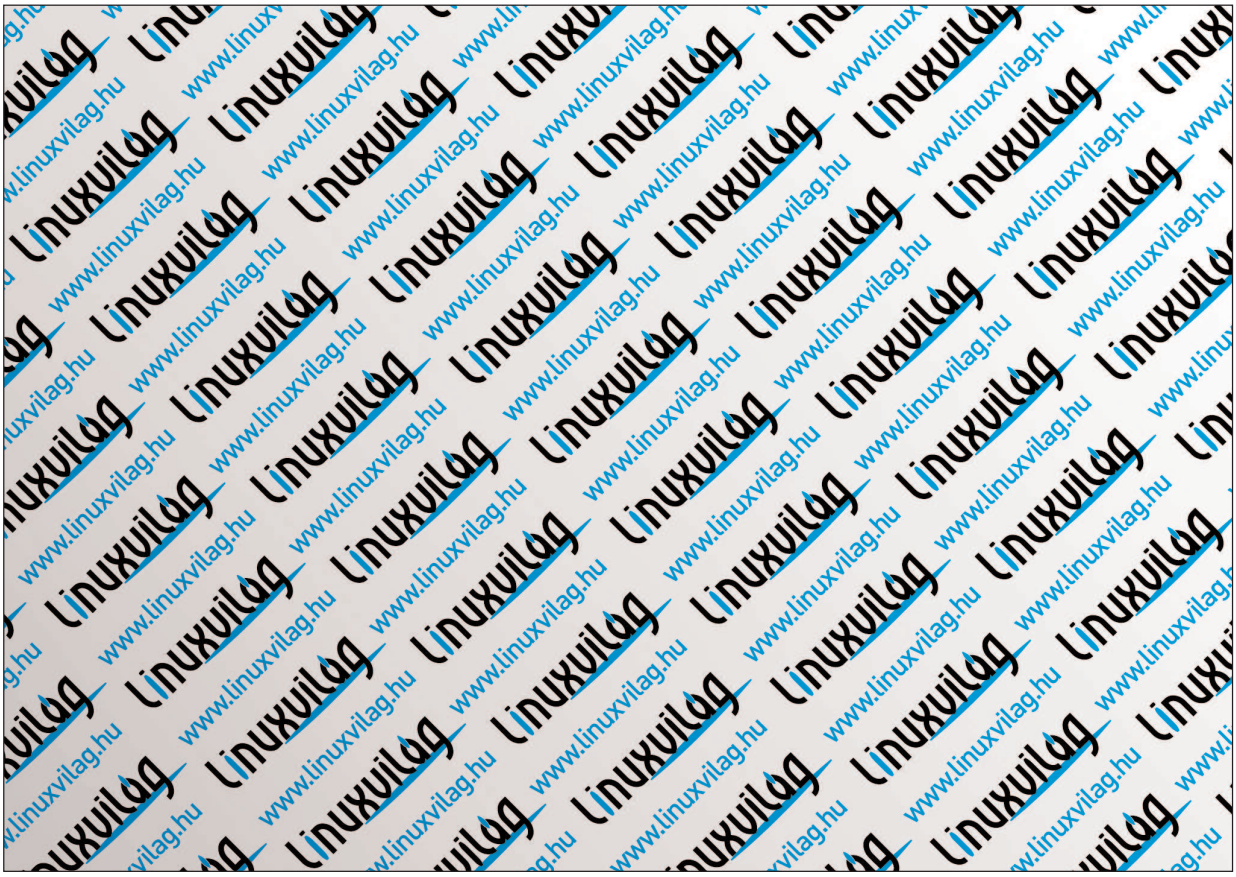
Köszönetnyilvánítás

Köszönettel tartozom *C. Surestnek* a cikk előkészítéséhez nyújtott segítségéért.

Linux Journal 2005. április, 132. szám

A cikk forrásai: ➲ *www.linuxjournal.com/article/8064*

Dr B. Thangaraju (bt_raju@vsnl.net) fizikából szerzett PhD fokozatot, és öt évig kutató munkatársként dolgozott az Indiai Tudományos Intézetnél. Jelenleg műszaki vezető Indiában, a Wipro Technologies Talent Transformation, Embedded Systems Focus csoportjánál. Munkája során a Linux belső világával, a rendszermaggal, illesztőprogramokkal, valamint beágyazott és valós idejű Linuxokkal foglalkozik.



Miért és hogyan használjunk Netlink Socket-eket

Használjuk kétirányú, alakítható módszert a rendszermag és a felhasználói tér közötti adatátadásra.

A rendszermag fejlesztése és karbantartása elég komoly feladat, ezért csak a legszükségesebb és teljesítmény szempontból legfontosabb kódok kerülnek a magba. Más részek, mint a ***GUI***, kezelési és vezérlő kód, általában felhasználói térben futó programként készülnek. Elég általános gyakorlat ***Linux*** alatt, hogy egy bizonyos képességet megvalósításkor felosztanak a rendszermag és a felhasználói tér között. Most már csak az a kérdés, hogy a rendszermag kód és a felhasználói tér hogyan kommunikálnak egymás között?
A válasz a rendszermag és a felhasználói tér közötti különféle ***IPC*** módszerekben rejlik. Ilyen a rendszerhívás, az ***ioctl***, a ***proc*** fájlrendszer vagy a ***netlink socket***. Ez a cikk a ***netlink sockettel*** foglalkozik és megpróbál rávilágítani ennek a hálózatbarát ***IPC***-nek az előnyeire.

Bevezetés

A ***netlink socket*** egy különleges ***IPC***, melyet a rendszermag és a felhasználói térbeli folyamatok közötti információcserére használunk. Teljes kétirányú kommunikációs csatornát létesít a két rész között ahol a felhasználói folyamatok szabványos ***socket API***-t a rendszermag modulok pedig egy különleges ***API***-t használnak. A ***netlink socket*** az `AF_NETLINK` címcsaládot használja szemben a ***TCP/IP*** socketek által használt `AF_INET` címekkel. Minden ***netlink socket*** képesség a ***include/linux/netlink.h*** rendszermag fejléc fájlban definiálja saját protokolltípusát.
Az alábbiakban bemutatunk a ***netlink socket*** által támogatott néhány képességet és protokolltípust:

- `NETLINK_ROUTE`: a felhasználói tér útvonalválasztó démonai között használt kommunikációs csatorna. Ilyen például a ***BGP***, ***OSPF***, ***RIP*** és a rendszermag csomagtovábbító modul. A felhasználói tér útvonalválasztó démonai a rendszermag útvonalválasztó tábláját frissítik ezzel a netlink protokolltípussal.
- `NETLINK_FIREWALL`: Az ***IPv4*** tűzfal kód által küldött csomagokat fogadja.
- `NETLINK_NFLOG`: a felhasználói tér ***iptable*** kezelő eszközei és rendszermag tér ***Netfilter*** modulja közötti kommunikációs csatorna.
- `NETLINK_ARPD`: az ***arp*** tábla kezelése felhasználói térből.

De miért használnak a fentiek ***netlink*** kapcsolatot rendszerhívás, ***ioctl*** vagy ***proc*** fájlrendszer helyett a felhasználói és a rendszermag világ közötti kommunikációhoz? Az új képességekhez tartozó rendszerhívások, ***ioctl***-ek vagy ***proc*** fájlok elkészítése egyáltalán nem egyszerű; sőt azt kockáztatjuk, hogy belerondítunk a rendszermagba és ezzel az egész rendszer stabilitását veszélyeztetjük. A ***netlink socket*** viszont egyszerű: egyetlen konstanst, a protokoll típust kell a ***netlink.h***-hoz fűznünk. Ezáltal a rendszermag modul és az alkalmazás egy ***socket***-szerű API-n keresztül máris képes a kapcsolattartásra.
A ***netlink*** aszinkron jellegű, ugyanis, akárcsak a a többi ***socket API***, egy ***socket*** sort használ az üzenet rohamok csillapítására. A ***netlink*** üzenetet küldő rendszerhívás az üzenetet a fogadó üzenetsorába helyezi, majd meghívja a fogadó átvétel kezelőjét. A fogadó az ***átvételkezelő környezetben (context)***futva eldöntheti, hogy azonnal feldolgozza-e az üzenetet, vagy a sorban hagyva inkább később, más környezetben dolgozza azt fel. A ***netlink***-el ellentétben, a rendszerhívások szinkron feldolgozást igényelnek. Következésképpen, ha a felhasználói térből rendszerhívással adunk át egy üzenetet a rendszermagnak, a rendszermag ütemezési felbontására is hatással lehetünk amennyiben az üzenet feldolgozása kellően hosszú ideig tart.
A rendszermagban a rendszerhívásokat megvalósító kód fordítási időben kerül a magba; ezért rendszerhívás kódot nem tehetünk betölthető modulba, pedig a legtöbb eszközvezérlő ezen az elven alapszik. A ***netlink socket*** esetében nincs fordítási idejű függőség a ***Linux netlink*** magja és a betölthető modulokban található ***netlink*** alkalmazások között. A ***netlink socket*** támogatja a ***multicasot***, ez újabb előnyt jelent a rendszerhívásokkal, ***ioctl*** és ***proc*** megoldásokkal szemben. A folyamat ***multicast*** üzenetet küldhet egy ***netlink*** csoportcímre, melyre tetszőleges számú más folyamat hallgathat. Ez a rendszermagból a felhasználói térbe irányuló majdnem tökéletes eseményelosztási lehetőséget biztosít számunkra.
A rendszerhívások és az ***ioctl IPC***-k egyirányúak abban az értelemben, hogy ilyen ***IPC***-t csak felhasználói térben futó alkalmazások kezdeményezhetnek. De mit tegyünk ha egy rendszermag modul szeretne sürgős üzenetet küldeni a felhasználói tér alkalmazásainak? Ezekkel az ***IPC***-kel ezt nem tudjuk

közvetlenül megvalósítani. Az alkalmazásnak általában adott időnként le kell kérdeznie a rendszermagot, hogy tudomást szerezzen az állapotváltozásokról, azonban az intenzív lekérdezés erőforrásigényes. A *netlink* viszont ügyesen megoldja ezt a problémát, hiszen lehetővé teszi, hogy a rendszermag is küldhessen üzeneteket. Ezért mondjuk, hogy a *netlink socket* kétirányú (duplex) képességekkel rendelkezik.
Végül a *netlin socket BSD socket* jellegű *API*-t nyújt amit a programfejlesztő közösség kiválóan ismer. Következésképpen a képzési költség jóval alacsonyabb mint egy meglehetősen titokzatos rendszerhívás *API*-t vagy *ioctlt* használnánk.

Összehasonlítás a BSD útvonalválasztó sockettel

A *BSD TCP/IP* verem megvalósításban van egy különleges *socket*, amelyet *routing socketnek* neveznek. Ennek címcsaládja az `AF_ROUTE`, protokollcsaládja a `PF_ROUTE` *socket* típusa pedig a `SOCK_RAW`. *BSD* alatt a folyamatok arra használják az útvonalválasztó *socketet*, hogy a rendszermag útvonalválasztó táblájához útvonalakat fűzzenek vagy távolítsanak el. *Linux* alatt, az útvonalválasztó *socketnek* a *netlink socket* `NETLINK_ROUTE` protokolltípusa felel meg. A *netlink socket* képességkészlete tehát a *BSD* útvonalválasztó *socket* képességeinek bővített halmaza.

Netlink Socket API

A felhasználó alkalmazások a szabványos *socket API* (`socket()`, `sendmsg()`, `recvmsg()` és `close()`) segítségével érhetik el a *netlink socketet*. Az *API* részletes meghatározásait a kézikönyvoldalakon találjuk. Itt most csak azzal foglalkozunk, hogyan válasszunk paramétereket az *API*-hoz a *netlink socket* használata során. Az *API* ismerős kell legyen mindenkinek, aki már készített hagyományos *TCP/IP socketet* használó hálózati alkalmazást. A `socket()` segítségével a következőképpen készíthetünk új *socketet*:

```
int socket(int domain, int type, int protocol)
```

A *socket tartomány (címcsalád)* jelen esetben az `AF_NETLINK`, a *socket* típus pedig vagy `SOCK_RAW` vagy `SOCK_DGRAM` lesz, ugyanis a `netlink datagram`-jellegű szolgáltatás.
A `protocol` (protokolltípus) határozza meg, hogy melyik *netlink* képességhez használjuk a *socketet*. Az előre definiált netlink protokolltípusok közül néhány: `NETLINK_ROUTE`, `NETLINK_FIREWALL`, `NETLINK_ARPD`, `NETLINK_ROUTE6` és a `NETLINK_IP6_FW`. Saját magunk is könnyen felvehetünk új protokolltípusokat.
Minden *netlink* protokolltípushoz maximum 32 *multicast* csoportot adhatunk meg. Minden *multicast* csoportot az `1<<i` bitmaszk képviseli, ahol `0<=i<=31`. Ez rendkívül hasznos lehet ha azonos képességen dolgozó folyamatcsoport és a rendszermag koordinálását kell megoldanunk. A *multicast netlink* üzenetek küldésével csökkenthetjük a szükséges rendszerhívások számát és megkímélhetjük az alkalmazásokat, hogy a *multicast* csoporttagságukkal bajlódjanak.

bind()

Akárcsak a *TCP/IP* socket, a *netlink* `bind()` *API* is *helyi (forrás) socket* címeket rendel a megnyitott *socketekhez*.
A *netlink* címszerkezete a következő alakú:

```
struct sockaddr_nl
{
 sa_family_t    nl_family; /* AF_NETLINK   */
 unsigned short nl_pad;    /* zero         */
 __u32          nl_pid;    /* folyamat pid */
 __u32          nl_groups; /* mcast csoportmaszk */
} nladdr;
```

A `bind()` használata során a `sockaddr_nl` `nl_pid` mezőjében elhelyezhetjük a hívó folyamat saját azonosítóját.
Így itt az `nl_pid` szolgál a `netlink socket` helyi címeként.
Az alkalmazás felelőssége, hogy az `nl_pid`-et egyedi 32-bites egész számmal töltse fel:

```
NL_PID Formula 1:  nl_pid = getpid();
```

Az 1. formula az alkalmazás folyamat azonosítóját használja `nl_pid` értéknek, amely teljesen természetes választás, feltéve, hogy az adott *netlink* protokolltípus esetében a folyamatnak csak egyetlen *netlink socketre* van szüksége.
Azokban az esetekben, amikor ugyanazon folyamat különféle szálai azonos *netlink* protokollba tartozó, de egyedi *netlink socketeket* nyitnak meg, a 2. formulát használhatjuk az `nl_pid` létrehozására:

```
NL_PID Formula 2: pthread_self() << 16 | getpid();
```

Ezzel a módszerrel, egyazon folyamat különféle *pthreads* szálai azonos *netlink* protokoll típushoz tartozó de egyedi *netlink socketet* hozhatnak létre. Tulajdonképpen akár egyetlen *pthread* is nyithat több azonos protokollba tartozó *netlink socketet*. A fejlesztőknek persze kicsit kreatívabbnak kell lenniük az `nl_pid` értékek készítésekor, de úgy gondolom ezt nem igazán tekinthetjük szokásos felhasználásnak.
Ha az alkalmazás fogadni szeretné az adott *multicast csoportnak* küldött *netlink* üzeneteket egy adott protokolltípus esetén, az összes vonatkozó *mulicast csoport* azonosítóját össze-`OR`-olva készítheti el az `sockaddr_nl` `nl_groups` mezőjét. Egyébként az `nl_groups` értéke nulla, így az alkalmazás kizárólag a neki szánt és a protokolltípushoz tartozó „unicast” üzeneteket kapja meg. Az nladdr kitöltése után a következő formában *bind*-elünk:

```
bind(fd, (struct sockaddr*)&nladdr,
sizeof(nladdr));
```

Netlink üzenet küldése

Amikor a rendszermagnak vagy egy más felhasználói folyamatoknak *netlink* üzenet küldünk, cél címként egy másik `struct sockaddr_nl` `nladdr` értéket kell megadnunk, éppen úgy ahogyan *UDP* csomagot küldenénk a `sendmsg()`-el. Ha az üzenetet a rendszermagnak szánjuk, az `nl_pid` és az `nl_groups` értéke 0.
Ha az üzenet egy másik folyamatnak szánt unicast üzenet, az `nl_pid` a folyamat `pid` azonosítója az `nl_groups` pedig 0, feltételezve, hogy rendszerünkben az 1. formulát használjuk.
Amennyiben az üzenet egy vagy több *multicast csoportnak* szánt *multicast üzenet*, valamennyi *multicast csoportot* össze-`OR`-olva képezzük az `nl_groups` mezőbe írandó maszkot. Ezek után a *netlink* címet beírhatjuk a `sendmsg()` *API*-hoz tartozó `struct msghdr msg` szerkezetbe:

```
struct msghdr msg;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);
```

A ***netlink socketnek*** saját üzenetfejlécre is szüksége van. Ezzel biztosítjuk a közös alapot valamennyi protokolltípus ***netlink*** üzenetei számára.
Minthogy a ***Linux*** rendszermag ***netlink*** magja a következő fejlécet keresi minden ***netlink*** üzenetben, az alkalmazásnak ezt a fejlécet minden elküldött ***netlink*** üzenetben át kell adnia:

```
struct nlmsghdr
{
  __u32 nlmsg_len;   /* Üzenet hossza */
  __u16 nlmsg_type;  /* Üzenettípus */
  __u16 nlmsg_flags; /* Kiegészítő jelek */
  __u32 nlmsg_seq;   /* Sorszám */
  __u32 nlmsg_pid;   /* Küldő folyamat PID */
};
```

Az `nlmsg_len` elemet a ***netlink*** üzenet teljes hosszával kell feltölteni, beleértve a fejlécet is, és kötelezően meg kell adni a ***netlink*** magnak. Az `nlmsg_type` értéket az alkalmazások használhatják és a ***netlink*** mag számár nem bír jelentőséggel. Az `nlmsg_flags` segítségével további üzenetvezérlést hajthatunk végre; ezt a ***netlink*** mag beolvassa és frissíti. Az `nlmsg_seq` és `nlmsg_pid` értékeket az alkalmazások használják az üzenetek nyomon követésére, és a ***netlink*** mag számára szintén lényegtelenek.
A ***netlink*** üzenet tehát a `nlmsghdr` fejlécből és az üzenet adataiból áll. Miután bevittük az üzenetet, az `nlh` mutató által megadott pufferbe kerül. Az üzenetet a `struct msghdr msg` szerkezetnek is elküldhetjük:

```
struct iovec iov;
iov.iov_base = (void *)nlh;
iov.iov_len = nlh->nlmsg_len;
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
```

A fenti lépéseket követően, a `sendmsg()` meghívásával kilőhetjük a ***netlink üzenetet***:
`sendmsg(fd, &msg, 0);`

Netlink üzenetek fogadása

A fogadó alkalmazásnak fenn kell tartania egy kellően nagy puffert amelyben az üzenet fejléce és az üzenet tartalma elfér. Ezek után az alább látható módon feltölti a `struct msghdr msg` szerkezetet majd a szabványos `recvmsg()` hívással fogadja a ***netlink*** üzenetet, feltételezve, hogy az `nlh` a bufferünkre mutat:

```
struct sockaddr_nl nladdr;
struct msghdr msg;
struct iovec iov;
iov.iov_base = (void *)nlh;
iov.iov_len = MAX_NL_MSG_LEN;
msg.msg_name = (void *)&(nladdr);
msg.msg_namelen = sizeof(nladdr);
msg.msg_iov = &iov;
msg.msg_iovlen = 1;
recvmsg(fd, &msg, 0);
```

Miután az üzenet helyesen megérkezett, az `nlh` az éppen megérkezett ***netlink*** üzenet fejlécére kell mutasson. A fogadott üzenet célcímét az `nladdr` tartalmazza, amely a ***pid***-ből és az üzenet címzettjeként megadott ***multicast*** csoportból áll. A ***netlink.h***-ban definiált `NLMSG_DATA(nlh)`, a ***netlink*** üzenet tartalmára mutat. A `close(fd)` hívás lezárja az `fd` leíró által megadott ***netlink socketet***.

Netlink API a rendszermag térben

A rendszermag térben futó ***netlink API***-t a rendszermag ***net/core/af_netlink.c netlink*** magja kezeli. A rendszermag oldaláról nézve, az ***API*** eltérő a felhasználó térbeli ***API***-tól. Az ***API***-t a rendszermag modulok használhatják a ***netlink socketek*** elérésre és a felhasználói alkalmazásokkal való párbeszédhez. Amennyiben nem a már meglévő ***netlink socket*** protokoll típusokkal dolgozunk, saját protokolltípusunkat állandóként fel kell vennünk a ***netlink.h*** állományba. Például, a tesztelési céllal készülő protokollunk felvételéhez a következő sort kell a ***netlink.h***-ba illesztenünk:
`#define NETLINK_TEST  17`

Ezt követően a felvett protokolltípusra a ***Linux*** rendszermagban bárhol hivatkozhatunk.
A felhasználói térben a `socket()` függvényt hívtuk a ***netlink socket*** készítésekor, kerneltérben azonban, a következő ***API***-t indítjuk:

```
struct sock *
netlink_kernel_create(int unit,
          void (*input)(struct sock *sk, int
          ➥len));
```

A `unit` paraméter nem más mint a ***netlink*** protokolltípus, azaz például a `NETLINK_TEST`. Az `input` függvénymutató lesz az a visszahívott függvény, amelyre az üzenetek érkeznek a ***netlink socketünkben***.
Miután a rendszermag létrehozta a `NETLINK_TEST` protokollhoz tartozó ***netlink socketet***, valahányszor a felhasználói térből egy `NETLINK_TEST` protokoll típusú üzenet érkezik a rendszermaghoz, a `netlink_kernel_create()`-el regisztrált az `input()` visszahívott függvény indul el. Nézzünk egy példát a visszahívott függvény megvalósítására:

```
void input (struct sock *sk, int len)
{
 struct sk_buff *skb;
 struct nlmsghdr *nlh = NULL;
 u8 *payload = NULL;
 while ((skb = skb_dequeue(&sk->receive_queue))
       != NULL) {
 /* feldolgozzuk a skb->data által megadott netlink
 ➥üzenetet*/
 nlh = (struct nlmsghdr *)skb->data;
 payload = NLMSG_DATA(nlh);
 /* Feldolgozzuk az üzenetet nlh-val jelölt
 ➥fejlécét
  * és a payload által jelölt tartalmát
  */
 }
}
```

Ez az input() függvény indul majd el a küldő folyamat által elindított sendmsg() rendszerhívás környezetében. A *netlink* üzenetet az input() függvényben is fel lehet dolgozni, feltéve, hogy elég gyors. Amikor a *netlink* üzenet feldolgozása hosszú időt venne igénybe, jobb ha az input() függvényen kívül helyezzük el, hogy ne blokkoljuk a többi rendszerhívás belépést a rendszermagba. Egy külön erre a célra fenntartott rendszermag szálat használunk a következő lépések végrehajtására. Végrehajtjuk az skb = skb_recv_datagram(nl_sk) kifejezést, ahol az nl_sk netlink_kernel_create()-től visszakapott *netlink socket*. Aztán feldolgozzuk az skb->data által mutatott *netlink üzenetet*.
Ez a rendszermag szál mindaddig alszik, amíg nincs *netlink üzenet* az nl_sk szerkezetben. Így aztán az input() visszahívott függvényünkben csak fel kell ébresztenünk az alvó rendszermag szálat, valahogy így:

```
void input (struct sock *sk, int len)
{
  wake_up_interruptible(sk->sleep);
}
```

Ez egy méretezhetőbb megoldás a felhasználói tér és a rendszermag között. Emellett a környezetváltások felbontását is javítja.

Netlink üzenetek küldése a rendszermagból

Akárcsak a felhasználói térben, a *netlink üzenet* kiküldésekor be kell állítanunk a forrás és a cél *netlink* címét. Feltételezve, hogy az elküldendő üzenetet tartalmazó *socket buffer* neve sk_buff *skb, a helyi címet a következőképpen állíthatjuk be:

```
NETLINK_CB(skb).groups = local_groups;
NETLINK_CB(skb).pid = 0;    /* a rendszermagból*/
```

A cél címét pedig így adjuk meg:

```
NETLINK_CB(skb).dst_groups = dst_groups;
NETLINK_CB(skb).dst_pid = dst_pid;
```

Az ilyen információt nem tároljuk az skb->data elemben. Ehelyett az skb socket buffer netlink vezérlőblokkjába kerül.
Az *unicast* üzenet elküldéséhez a következőket használjuk:

```
int
netlink_unicast(struct sock *ssk, struct sk_buff
                *skb, u32 pid, int nonblock);
```

Itt az ssk a netlink_kernel_create(), által visszaadott *netlink socket* az skb->data az elküldendő *netlink* üzenetre mutat a pid a fogadó alkalmazás *pid*-je, feltételezve, hogy az 1. NLPID formulát alkalmaztuk. A nonblock jelzi, hogy ha a fogadó puffer nem érhető el az *API* blokkoljon-e vagy azonnal térjen vissza hibával.
Multicast üzeneteket is küldhetünk. A következő *API* a *netlink üzenetet* kézbesít a *pid* által meghatározott folyamatnak és a *group (csoport)* által megadott csoportoknak:

```
void
netlink_broadcast(struct sock *ssk, struct sk_buff
        *skb, u32 pid, u32 group, int allocation);
```

A group a fogadó csoportok OR művelettel egyesített bitmaszkja. Az allocation a rendszermag memória foglalási típusa. Általában GFP_ATOMIC típust adunk meg ha megszakítás környezetről van szó, egyébként pedig GFP_KERNEL-t. Ez azért lényeges, mert az *API* egy vagy több puffert is kénytelen lehet lefoglalni amikor lemásolja a multicast üzenetet.

Netlink Socket lezárása a rendszermagból

A netlink_kernel_create() által visszaadott struct sock *nl_sk segítségével meghívhatjuk a következő rendszermag *API*-t, amely a rendszermagban lezárja a *netlink socketet*:

```
sock_release(nl_sk->socket);
```

Eddig csak a legminimálisabb keretrendszer kódot mutattuk be a *netlink* programozás demonstrálására. Most használjuk a NETLINK_TEST *netlink* protokoll típust és tételezzük fel, hogy már hozzá van adva a rendszermag fejléchez. Az itt bemutatott rendszermag modul kód csak a *netlink* vonatkozású részekkel foglalkozik, tehát mindezt még be kellene illeszteni egy rendszermag modul vázba, amit rengeteg egyéb ismertetőben megtalálunk.

A rendszermag és az alkalmazás közötti Unicast kommunikáció

Példánkban, a felhasználói tér folyamatai küldenek üzeneteket a rendszermagnak, a rendszermag modul pedig visszajuttatja az üzenetet a küldő folyamatnak. Íme a felhasználói tér kódja:

```
#include <sys/socket.h>
#include <linux/netlink.h>
#define MAX_PAYLOAD 1024  /* maximális tartalom
➥mérete*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
int sock_fd;
void main() {
 sock_fd = socket(PF_NETLINK,
SOCK_RAW,NETLINK_TEST);
 memset(&src_addr, 0, sizeof(src_addr));
 src__addr.nl_family = AF_NETLINK;
 src_addr.nl_pid = getpid();  /* saját pid */
 src_addr.nl_groups = 0;  /* nincs mcast csoportban
 ➥*/
 bind(sock_fd, (struct sockaddr*)&src_addr,
      sizeof(src_addr));
 memset(&dest_addr, 0, sizeof(dest_addr));
 dest_addr.nl_family = AF_NETLINK;
 dest_addr.nl_pid = 0;   /* A Linux rendszermagnak
 ➥*/
 dest_addr.nl_groups = 0; /* unicast */
 nlh=(struct nlmsghdr *)malloc(
```

```
                                NLMSG_SPACE(MAX_PAYLOAD));
 /* kitöltjük a netlink üzenetfejlécet */
 nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
 nlh->nlmsg_pid = getpid();  /* self pid */
 nlh->nlmsg_flags = 0;
 /* Feltöltjük a netlink üzenetet tartalommal */
 strcpy(NLMSG_DATA(nlh), "Hello you!");
 iov.iov_base = (void *)nlh;
 iov.iov_len = nlh->nlmsg_len;
 msg.msg_name = (void *)&dest_addr;
 msg.msg_namelen = sizeof(dest_addr);
 msg.msg_iov = &iov;
 msg.msg_iovlen = 1;
 sendmsg(fd, &msg, 0);
 /* Üzenet olvasása a rendszermagtól */
 memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));
 recvmsg(fd, &msg, 0);
 printf(" Received message payload: %s\n",
        NLMSG_DATA(nlh));

 /* Lezárjuk a netlink socketet */
 close(sock_fd);
}
```

Most nézzük a rendszermag kódot:

```
struct sock *nl_sk = NULL;
void nl_data_ready (struct sock *sk, int len)
{
  wake_up_interruptible(sk->sleep);
}
void netlink_test() {
 struct sk_buff *skb = NULL;
 struct nlmsghdr *nlh = NULL;
 int err;
 u32 pid;
 nl_sk = netlink_kernel_create(NETLINK_TEST,
                                    nl_data_ready);
 /* várunk amig üzenet érkezik a felhasználói
 ➥térből */
 skb = skb_recv_datagram(nl_sk, 0, 0, &err);
 nlh = (struct nlmsghdr *)skb->data;
 printk("%s: received netlink message
payload:%s\n",
        __FUNCTION__, NLMSG_DATA(nlh));
 pid = nlh->nlmsg_pid; /* küldő folyamat pid-je */
 NETLINK_CB(skb).groups = 0; /* nincs mcast
 ➥csoportban */
 NETLINK_CB(skb).pid = 0;      /* a kernelből */
 NETLINK_CB(skb).dst_pid = pid;
 NETLINK_CB(skb).dst_groups = 0;  /* unicast */
 netlink_unicast(nl_sk, skb, pid, MSG_DONTWAIT);
 sock_release(nl_sk->socket);
}
```

Miután betöltöttük a fenti rendszermag kódot tartalmazó modult és lefuttatjuk az végrehajtható állományt, a felhasználói térben futó programunk következő üzenetet fogja kiírni:

```
Received message payload: Hello you!
```

A dmesg kimenetében pedig a következőknek kell megjelennie:

```
netlink_test: received netlink message payload:
Hello you!
```

A rendszermag és az alkalmazás közötti multicast kommunikáció

Ebben a példában két felhasználói térben futó alkalmazás hallgat ugyanarra a ***netlink multicast*** csoportra. A rendszermag modul a ***netlink socketen*** keresztül egy üzenetet dob a ***multicast*** csoportnak, amelyet valamennyi alkalmazás megkap. Nézzük a felhasználói tér kódját:

```
#include <sys/socket.h>
#include <linux/netlink.h>
#define MAX_PAYLOAD 1024  /* maximális tartalom
➥mérete*/
struct sockaddr_nl src_addr, dest_addr;
struct nlmsghdr *nlh = NULL;
struct iovec iov;
int sock_fd;
void main() {
 sock_fd=socket(PF_NETLINK, SOCK_RAW,
NETLINK_TEST);
 memset(&src_addr, 0, sizeof(local_addr));
 src_addr.nl_family = AF_NETLINK;
 src_addr.nl_pid = getpid();  /* saját pid */
 /* interested in group 1<<0 */
 src_addr.nl_groups = 1;
 bind(sock_fd, (struct sockaddr*)&src_addr,
      sizeof(src_addr));
 memset(&dest_addr, 0, sizeof(dest_addr));
 nlh = (struct nlmsghdr *)malloc(

NLMSG_SPACE(MAX_PAYLOAD));
 memset(nlh, 0, NLMSG_SPACE(MAX_PAYLOAD));

 iov.iov_base = (void *)nlh;
 iov.iov_len = NLMSG_SPACE(MAX_PAYLOAD);
 msg.msg_name = (void *)&dest_addr;
 msg.msg_namelen = sizeof(dest_addr);
 msg.msg_iov = &iov;
 msg.msg_iovlen = 1;
 printf("Waiting for message from kernel\n");
 /* üzenet beolvasása a rendszermagból */
 recvmsg(fd, &msg, 0);
 printf(" Received message payload: %s\n",
        NLMSG_DATA(nlh));
 close(sock_fd);
}
```

A rendszermag kód a következőképpen néz ki:

```
#define MAX_PAYLOAD 1024
struct sock *nl_sk = NULL;
void netlink_test() {
 sturct sk_buff *skb = NULL;
 struct nlmsghdr *nlh;
 int err;
```

```
 nl_sk = netlink_kernel_create(NETLINK_TEST,
                               nl_data_ready);

skb=alloc_skb(NLMSG_SPACE(MAX_PAYLOAD),
➥GFP_KERNEL);
 nlh = (struct nlmsghdr *)skb->data;
 nlh->nlmsg_len = NLMSG_SPACE(MAX_PAYLOAD);
 nlh->nlmsg_pid = 0;  /* a rendszermagból */
 nlh->nlmsg_flags = 0;
 strcpy(NLMSG_DATA(nlh), "Greeting from kernel!");
 /* sender is in group 1<<0 */
 NETLINK_CB(skb).groups = 1;
 NETLINK_CB(skb).pid = 0;  /* a rendszermagból */
 NETLINK_CB(skb).dst_pid = 0;  /* multicast */
 /* to mcast group 1<<0 */
 NETLINK_CB(skb).dst_groups = 1;
 /*multicast üzenet minden figyelő folyamatnak */
 netlink_broadcast(nl_sk, skb, 0, 1, GFP_KERNEL);
 sock_release(nl_sk->socket);
}
```

Tételezzük fel, hogy a végrehajtható állományt `nl_recv` néven fordítottuk le. Indítsuk el két példányban az `nl_recv` programot:

```
./nl_recv &
Waiting for message from kernel
./nl_recv &
Waiting for message from kernel
```

Aztán, amikor betöltöttük a rendszermag térben futó kódot végrehajtó modult, az nl_recv mindkét példánya a következő üzenetet kapja:

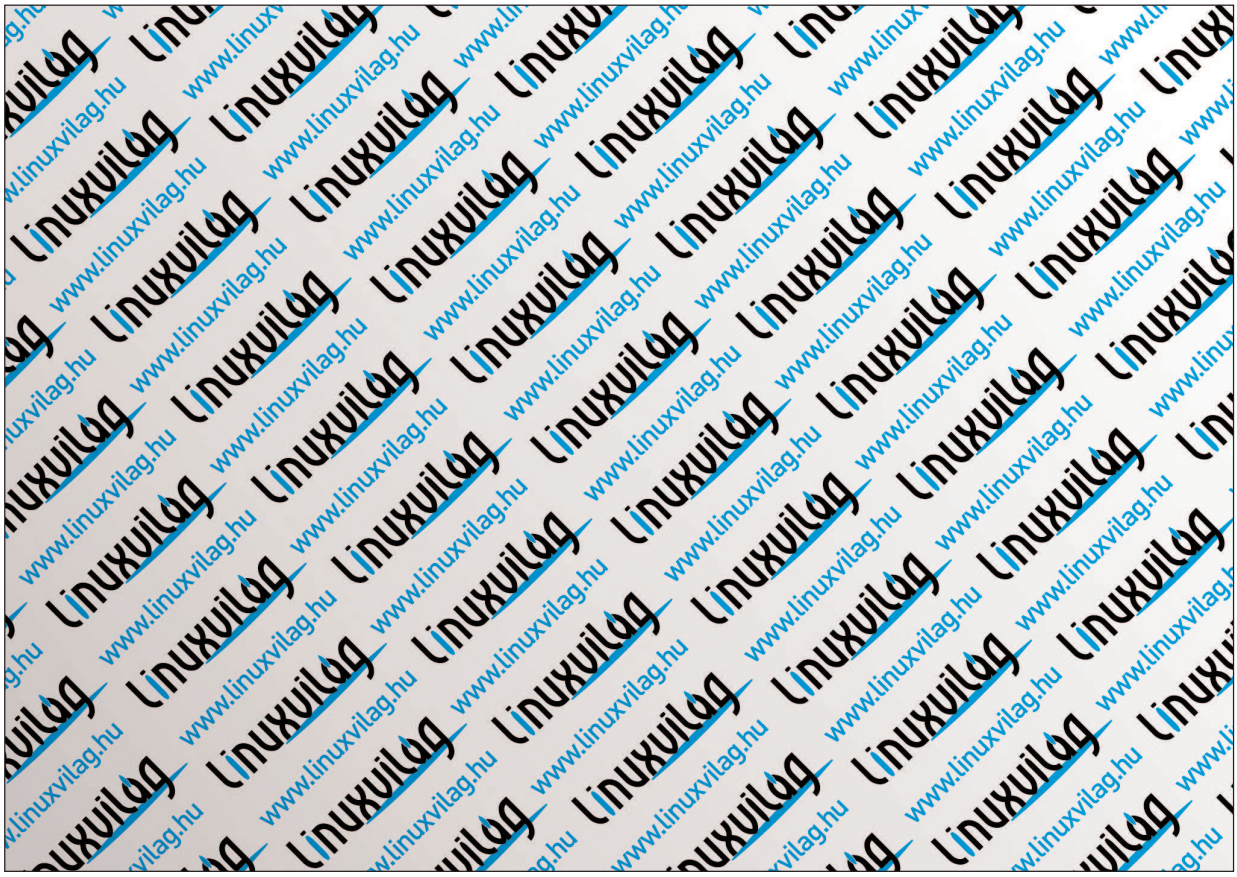
```
Received message payload: Greeting from kernel!
Received message payload: Greeting from kernel!
```

Összefoglalás

A ***Netlink socket*** rugalmas kommunikációs felületet biztosít a felhasználói tér alkalmazásai és a rendszermag modulok között. Könnyen kezelhető ***socket API***-val rendelkezik az alkalmazás és a rendszermag oldalán egyaránt. Olyan fejlett kapcsolattartó képességekkel bír, mint a full-duplex, pufferelt I/O, és aszinkron kommunikáció, amelyek más rendszermag/felhasználói-tér IPC-kből hiányoznak.

Linux Journal 2005. február, 130. szám

Kevin Kaichuan He
(hek_u5@yahoo.com)
A Solustek Corp vezető szoftvermérnöke. Jelenleg beágyazott rendszereken, eszközmeghajtókon, és hálózati protokoll projekteken dolgozik. Korábban mint a Cisco Systems vezető mérnöke és a Purdue Egyetem tudományos munkatársa szerzett tapasztalatokat. Szabadidejében szívesen készít digitális fotókat, játszik PS2-es játékokat és olvas.



Saját nyelvi felület készítése GCC-hez

Jó hír a nyelvtervezőknek hogy egyszerűbbé vált a nyelvi felületek készítése a GCC-hez.

A *GCC*, ez az elsőrangú ingyenes programfordító készlet, rengeteg változáson ment keresztül az utóbbi néhány évben. Az egyik ilyen változás, nevezetesen a ***tree-ssa*** ág beolvasztása lényegesen leegyszerűsítette az új *GCC* előtétek készítését.
A *GCC* mindig is két belső megvalósítással rendelkezett, a fákkal és az *RTL*-el. Az *RTL*, azaz regiszter átviteli nyelv, az alacsonyabb szintű megoldás, ezt használja a *GCC* a gépi kód készítésekor. Korábban minden optimalizálás az *RTL* szintjén folyt. A fák magasabb szintű megoldások. Hagyományosan ezek kevésbé dokumentáltak és kevésbé ismertek voltak mint az *RTL*.
A ***Diego Novillo*** vezette, *GCC* belső részeinek hosszú távú újraírását célzó ***tree-ssa*** projekt mindezt megváltoztatja.
A fák azóta sokat fejlődtek, igaz még mindig kicsit hiányosan dokumentáltak, viszont sok optimalizálás már a fa szinten megtörténik. A munka hasznos melléktermé a fa alapú nyelvek egyértelmű megfogalmazása: a *GENERIC*. Minden *GCC* nyelvi felület *GENERIC*-et készít, amelyet később egy másik, *GIMPLE* nevű alacsonyabb szintű fa alapú leírássá alakítunk, végül innen lépünk tovább a *RTL*-re. Ami ebből számunkra lényeges, hogy mostantól sokkal, de sokkal könnyebb nyelvi felületet írni a *GCC*-hez. Tulajdonképpen, most már akár az is képes *GCC* nyelvi felületet készíteni, aki egyáltalán nem ért az *RTL* megvalósításokhoz. Ez a cikk végigvezet bennünket azokon a lépéseken, melyekkel elkészíthetjük saját fordítóprogramunk *GCC* felületét. A cikkben leírtak a 2005-ben megjelent *GCC 4.0*-ás verzióra vonatkoznak.

A program ábrázolása

A mi szempontunkból a fordítás két részből áll, értelmezés és szemantikai vizsgálat, majd a kódgenerálás. A *GCC* a második részt elvégzi, így már csak az a kérdés hogyan tudjuk a legjobban megvalósítani az első fázist?
A hagyományos *GCC* nyelvi felületek, például a *C* és a *C++* felület, értelmezés közben készítik el a fát. Az ilyen felületek a nyelv specifikus szerkezetekhez saját fa-kódot adnak meg. Aztán amikor a szemantikai vizsgálat befejeződött, ezeket a fákat *GENERIC*-é alakítják, a magas szintű nyelvspecifikus fákat alacsonyabb szintű változatokkal helyettesítve. A megközelítés egyik előnye, hogy a nyelvspecifikus fák már amúgy is közel *GENERIC* alakúak. A szintcsökkentési fázissal gyakran elkerülhető, hogy túlságosan sok szemét keletkezzen.
A legfőbb probléma ezzel a megközelítéssel, hogy a fa dinamikus típusú. Elméletben ez még nem olyan nagy baj, hiszen rengeteg dinamikus típusú környezet létezik amelyeket hatékonyan használhatnak a fejlesztők, például ilyen a ***Lisp*** és a ***Python*** is. Ugyanakkor ezek teljes környezetek, a *GCC* erősen „makrósított" *C* kódjáról viszont nem lehet elmondani ugyanezeket az előnyöket.
Magam részéről azokat a megközelítéseket kedvelem, amelyek erősen típusos nyelvspecifikus programábrázolást más néven az úgynevezett ***„abstract syntax tree" (AST)*** megoldást alkalmazzák. Ezt a megközelítést alkalmazza az Ada felület és a Java programozási nyelvhez készült nyelvi felület újraírt változata, a *gcjx*.
A *gcjx C++* nyelven készült és a java programozási nyelv osztályait modellező osztály hierarchiával rendelkezik.
Ez a kód lényegében független a *GCC*-től és akár más célra is felhasználható. A *gcjx* esetében például a modellt levihetjük a *GENERIC* szintjére, de akár használhatjuk bájtkód vagy *JNI* fejlécfájlok előállítására is. Ezen kívül akár kódvizsgálatot is végezhetünk; a felület gyakorlatilag újrahasznosítható könyvtár lesz.
Ez a megközelítés az erősen típusos tervezés minden előnyét élvezi. A *GCC* esetében ez konkrétan annyit jelent, hogy könnyebben karbantartható és érthető programot kapunk. Az eredményül kapott felület és a *GCC* viszonylagos függetlensége azért is előnyös, mert a *GCC* igen gyorsan változik és ez a laza párosítás minimálisra csökkenti hiba lehetőségét.
A megközelítés nyilvánvaló hátránya hogy a fordítóprogramunknak valószínűleg több munkát kell végeznie mint amennyire szigorúan szüksége lenne, vagy több memóriát használ. A gyakorlatban ez nem tűnik igazán fontosnak.
Mielőtt belevágnánk a *GCC* felületkészítés részleteinek megismerésébe, nézzük meg milyen dokumentációkat és forrásfájlokat érdemes megismernünk. Tekintettel arra, hogy a *GCC* közösség nem tulajdonított túl nagy jelentőséget az egyszerűen elkészíthető nyelvi felületnek, néhány fontos dolog csak a forrásban van dokumentálva. Az itt felsorolt dokumentációk ***info*** lapokra és nem ***URL***-ekre hivatkoznak, hiszen a *GCC 4.0* még nem jött ki. Ezért aztán a weblapokon a korábbi verziókat láthatjuk. A legjobb amit tehetünk, hogy a *CVS*-ből letöltjük a *GCC*-t és átlapozzuk a forráskódot.

- *gcc/c.opt*: leírja a *C* családba tartozó nyelvi felületek parancssori kapcsolóit. És ami még fontosabb bemutatja az *.opt* állományok formátumát. Ilyet kell majd írnunk.

- *gcc* infó lap, ***Spec Files*** csomópont (***gcc/doc/invoke.texi*** forrásfájl): a *GCC* meghajtó által használt meghatározó mininyelvet írja le. Ilyen meghatározásokat fogunk készíteni, amely megmutatja a *GCC*-nek, hogyan kell meghívnia a felületünket.
- ***gccint*** infó lap, ***Front End*** csomópont (***gcc/doc/sourcebuild.texi*** forrásfájl): ismerteti hogyan integráljuk a felületünket a *GCC* fordítási folyamatába.
- ***gccint*** infó lap, ***Tree SSA*** csomópont (***gcc/doc/tree-ssa.texi*** forrásfájl): a *GENERIC* leírása.
- ***gcc/tree.def***, ***gcc/tree.h***: a fák néhány olyan jellemzője ami úgy tűnik kimaradt a dokumentációkból, ezeket a fájlokat elolvasva megismerhető. A ***tree.def*** definiálja a legtöbb fa kódot és tartalmaz néhány átfogó megjegyzést. A ***tree.h*** definiálja a fa szerkezetét, a sok elérési makrót valamint néhány függvényt, amelyek a különféle fák létrehozása során lehetnek segítségünkre.
- ***libcpp/include/line-map.h***: a sortérképet a *GCC* forráskód helyek tárolására használja. Ezt tetszés szerint használhatjuk a saját nyelvi felületünk megvalósításában. A ***gcjx*** nem használja. Ha nem is használjuk őket, a *GENERIC* szintjére lépéskor mindenképpen el kell készíteni azokat, ugyanis a sortérképben tárolt információkat használjuk a hibakeresési információk előállításához.
- ***gcc/errors.h***, ***gcc/diagnostic.h***: A *GCC* hibaformázási függvényeit definiálja ezeket tetszés szerint használhatjuk.
- ***gcc/gdbinit.in***: néhány *GDB* parancsot definiál amelyek jól jöhetnek ha hibát keresünk a *GCC*-ben. Például a `pt` parancs szöveges alakban rajzolja ki a fát. A *GCC* fordítási könyvtárában készül egy ***.gdbinit*** állomány, így ha hibát keresünk a makrók azonnal elérhetőek lesznek.
- ***gcc/langhooks.h***: a *GCC* a nyelvi horgok módszerének alkalmazásával teszi lehetővé a nyelvi felületeknek, hogy a *GCC* viselkedését befolyásolhassák. Minden felületnek saját ***langhooks*** struktúrát kell készítenie; Ezek a struktúrák lényegében függvénymutatókból állnak. A *GCC* közép és alsó rétege hívja meg ezeket a függvényeket ha nyelvfüggő döntetéseket kell hozni a fordítás folyamán. A ***langhooks*** szerkezetek időről időre változnak, azonban a *GCC* olyan módszerrel igényli a szerkezetek alaphelyzetbe állítását, hogy ezek a változások lényegében nem jelentenek problémát forráskód szinten. Néhány nyelvi hurok nem szabadon választható, így a felületünkhöz mindenképpen el kell készítenünk azokat. Mások ad hoc megoldások egy adott problémára. Például, a `can_use_bit_fields_p` horog kizárólag a ***gcj*** rendszer optimalizálási problémájának megoldása kedvéért került a rendszerbe.

Meghajtó készítése

Jelenleg a *GCC* megköveteli, hogy a felületünk fordításkor elérhető legyen. Nem tudunk olyan felületet készíteni amely egy már meglévő *GCC* telepítéshez csatlakozik. Ennél a lépésnél érdemes elolvasni a vonatkozó *GCC* kézikönyv oldalakat, hogy megtudjuk hogyan készítsük el a felülethez szükséges fordítási környezetet. Természetesen a legegyszerűbb módszer ha egy már meglévő nyelvi felület állományait másoljuk át és azt módosítjuk az igényeinknek megfelelően. Ezek után két fájlt készítünk el, amelyek segítenek felületünket a *GCC* meghajtóprogramba illeszteni. A ***lang-specs.h*** állomány a felületünket ismerteti a *GCC* meghajtó számára. Megmutatja a meghajtónak, hogy amikor ezt a kiterjesztést látja a parancssorban a *GCC* a mi felületünket kell meghívja. Ezen kívül az egyéb meghívandó programokkal kapcsolatos információkkal látja el a meghajtót, például kell-e assemblert futtatni a felületünk után, és hogyan adjunk át vagy módosítsunk bizonyos parancssori kapcsolókat. Ennek az állománynak elkészítése egy kis időbe telhet hiszen a leírásnak saját különös nyelve van. Más nyelvi felületek állományai sokat segíthetnek. A ***lang.opt*** a felületünkhöz tartozó parancssori kapcsolókat mutatják be. Ez az egyszerű szöveges állomány egyértelmű formátumot használ. Az egyszerű kapcsolókat például a figyelmeztető zászlókat elegendő a ***lang.opt*** állományban megadni és semmilyen további programozást nem igényelnek. a többi paramétert általunk írt nyelvi horgoknak kell kezelnie. Ezek után készítsük el a fordítás folyamatát vezérlő nyelvi horgokat. E csoport legfontosabb elemei:

- `init_options`: ez a felületünkhöz intézet első hívás a paraméterek feldolgozása előtt.
- `handle_option`: egy parancssori kapcsoló kezeléséhez meghívott horog.
- `post_options`: a parancssori feldolgozás végeztével hívódik meg. Ebben a nyelvi horogban kényelmesen meghatározhatjuk az értelmezendő állomány nevét.
- `init`: a `post_options` után hívódik meg és alaphelyzetbe állítja a felületünket.
- `finish`: A fordítás végén hívódik meg. Szükség esetén ezt használhatjuk a felület utómunkálatainak elvégzéséhez.
- `parse_file`: ez a nyelvi horog végzi el a bemeneti állományhoz tartozó teljes értelmezést, szemantikai vizsgálatot és a kódgenerálást. Lényegében itt történik a fordítási munka.

Alaphelyzetbe állítás

A *GCC* elvárja hogy nyelvi felületünk alaphelyzetbe állítsa magát. A *GCC* legtöbb része ön-előkészítő, de a különféle felületek igényeihez alkalmazkodva lehetőségünk van néhány fákkal kapcsolatos globális változó létrehozására nem típus módon. Azért azt javaslom ebbe ne nagyon mélyedjünk bele. Sokkal egyszerűbb szabványos módszerrel szabványos fa csomópontokat definiálni és saját neveket kitalálni a fákhoz amelyek mondjuk nyelvünk szabványos típusait tartalmazzák. Az alaphelyzetbe állítás során esetleg meghívhatjuk a `build_common_tree_nodes`, `set_sizetype` és `build_common_tree_nodes_2` függvényeket. A `set_sizetype` `size_t` belső megfelelőjének méretét állítja be; a legegyszerűbb ezt mindig `long_unsigned_type_node` értékre állítani. Más beállításokat is végezhetünk ebben a lépésben. A ***gcjx*** alaphelyzetbe állító kódja például különböző struktúráknak megfelelő típusokat hozunk létre amelyek a ***Java*** osztályok és metódusok leírásához szükségesek.

Fordítás GENERIC-re

A `parse_file` nyelvi horog meghívja a fordítónkat, hogy az elkészíthesse a belső adatszerkezeteinket. Feltételezve, hogy ez hiba nélkül lezajlott, a felületünk készen áll a *GENERIC* fák előállítására saját ***AST*** szerkezetünk alapján. A ***gcjx*** esetében mindez úgy történik, hogy egy különleges „vendég” ***API***-val végiglépkedünk az osztályhoz tartozó ***AST***-on. Az ***API GENERIC***-féle megvalósítása lépésenként építi fel a kód-

nak megfelelő fákat majd átadja a *GCC*-nek. A fák elkészítésének részletes ismertetése már meghaladná cikkünk kereteit. Az alábbi példák azonban bemutatják a három legnagyobb fa típust így megnézhetjük melyik hogyan is néz ki.

Type

Ez a fafajta a típusokat adja meg. Nézzünk egy példát a `gcjx`-ből a `char Java` típusra:

```
tree type_jchar = make_node (CHAR_TYPE);
TYPE_PRECISION (type_jchar) = 16;
fixup_unsigned_type (type_jchar);
```

A fák segítségével bármilyen típust le tudunk írni. Van például rekordokat, uniókat, mutatókat és különféle méretű egészeket leíró faszerkezet.

Decl

A `Decl` az értékadásnak felel meg, avagy más szavakkal egy objektumnak adott névnek. Például a forráskódban megadott helyi változó `decl` leírása:

```
tree local = build_decl (VAR_DECL, get_identifier
➥("variable_name"),
                         type_jchar);
```

A `decl`-ek különféle nevesített objektumokat jelenthetnek a programban: ***fordítási egységet (translation unit)***, függvényeket, mezőket, változókat, paramétereket, konstansokat, címkéket és típusokat. A típus `decl` a típus deklarációjának felel meg, szemben magával a típussal.

Expr

A programban előforduló különféle kifejezések leírásához sokféle *expr* fa áll rendelkezésünkre. Ezek hasonlóak a *C* kifejezésekhez, de bizonyos értelemben annál általánosabbak. A fák például nem tesznek különbséget az if utasítás és a feltételes kifejezés között - mindkettőt a `COND_EXPR` írja le, és az egyetlen különbség közöttük annyi, hogy az `if` utasítás `void` típusú. A következő kifejezés önmagát adja egy változóhoz:

```
tree addition = build2 (PLUS_EXPR, type_jchar,
local, local);
```

Az utasításoknak megfelelő fákat egy különleges közelítő *API* csatolja egymáshoz. Két utasítást (s1 és s2) a következőképpen csatolhatjuk egymáshoz:

```
tree result = alloc_stmt_list ();
tree_stmt_iterator out = tsi_start (result);
tsi_link_after (&out, s1, TSI_CONTINUE_LINKING);
tsi_link_after (&out, s2, TSI_CONTINUE_LINKING);
// most a  `result'-ba az utasítások listája került.
```

Más facsomópontok is léteznek; minden megtalálunk a *tree.def* állományban és a kézikönyvoldalon. A felületek saját fakódot is meghatározhatnak; azonban ha saját *AST*-nk van erre nem lesz szükségünk. A készülő program teljes szerkezete valószínűleg a `decl` fordítási egységre hasonlít majd, amely típusokat, változókat és függvényeket tartalmaz.

Átadás

Miután elkészítettük a függvényt, globális változót vagy típust leíró fát, amelyhez hibakeresési információt szeretnénk készíteni, a fát át kell adnunk a megfelelő függvénynek amely elvégzi a fordítás maradék részét. Jelenleg három ilyen függvény létezik: `rest_of_decl_compilation` a `decl` csomópont fordítását végzi el, a `cgraph_finalize_function` a függvények fordítását kezeli és a `rest_of_type_compilation` a típusok fordításával foglalkozik.

Hibakeresés

Bár a *GCC*-ben szép számmal találunk belső konzisztencia ellenőrzési pontokat, még így is elég könnyű összeomlást produkálni a felületünktől idegen kóddal. A legtöbb összeomlás esetén végiglépdelhetünk a vermen és megnézhetjük milyen fák voltak módosítva, amíg meg nem találjuk a hibás fakészítés okozta hibát. Ehhez a módszerhez meglepően kevés általános *GCC* ismeret is elegendő, így is hatékonyan tudunk hibákat keresni a kódban. A *GCC* rendelkezik néhány kézreálló hibakeresési lehetőséggel. A hibakeresőben meghívhatjuk a `debug_tree` függvényt, amely kinyomtatja a fát. Az `-fdump-tree` parancssori kapcsolócsaládal adott számú menet után is kiírathatjuk a fák értékét.

Tapasztalatok

A *gcjx* írása során szerzett tapasztalataim szerint az erősen típusos köztes megfelelő átalakítása fákká igen egyszerű volt. A *gcjx* fa háttérkódja, egy háttérkód a sok közül, a teljes fordító kódjának mintegy 10%-át teszi ki. Bár még befejezetlen körülbelül 6,000 kódsorból áll (nyers `wc -l` számlálás alapján) körülbelül akkora mint a bájtkód háttér. Ebből le lehet vonni azt a következtetést, hogy ha a fordítóprogramunk már készen van a *GCC*-hez kapcsolása már könnyedén megvalósítható. Tekintve, hogy a fák magas szintűek, egyetlen *RTL*-t sem néztem meg a felület készítése során és egyáltalán semmi időt nem kellett eltöltenem azzal, hogy processzor-specifikus problémákon törjem a fejemet. Amennyiben a nyelvnek amit választottál nincsenek különleges követelményei, veled is hasonló lesz a helyzet. A *gcjx* statikusan típusos *AST*-je könnyedén újrahasznosítható. Jelenleg négy háttér létezik és valószínűleg többet is írok később. Például nem lenne nehéz például olyan hátteret készíteni, amely a program kereszthivatkozásait menti adatbázisba. Vagy írhatnánk olyan hátteret amely végiglépked az *AST*-n és kikeresi a tipikus hibákat, a *FindBugs* programhoz hasonlóan. Ez az elgondolás még ösztönzőbb lehet olyan nyelvek esetében amelyek, a *Javaval* ellentétben, nem rendelkeznek gazdag analizáló készletekkel.

Jövőbeli elképzelések

A felületek írása természetesen lehetne még ennél is egyszerűbb. Például nincs szükség a *lang-specs.h* beillesztésére. Ehelyett a nyelvi felület telepíthetne egy leíróállományt, amit a *GCC* meghajtó induláskor beolvasna. Hasonlóképpen a *lang.opt* is elhagyható lenne. Egy kis munkával idővel talán az is elérhető, hogy a *GCC*-től függetlenül tudjunk nyelvi felületeket fordítani.

A cikk forrásai: ***www.linuxjournal.com/article/8138***

Linux Journal 2005. május, 133. szám

Tom Tromey

CGI programozás C nyelven

Az összetett webes feladatok elvégzését a CGI egyszerűségének megőrzése mellett is felgyorsíthatjuk. A rengeteg hasznos könyvtárnak köszönhetően a parancsfájl-készítő nyelvekről C-re áttérni nem is olyan nagy ugrás, mint azt gondolnánk.

A *Perl*, a *Python* és a *PHP* alkotja a *CGI* alkalmazásprogramozás szentháromságát. A boltok polcai roskadoznak az ezekről a nyelvekről szóló könyvektől, a számítástechnikai sajtó úton-útfélen velük foglalkozik, ahogy számos internetes fórumnak is ezek adják a témáját. Szembetűnő ugyanakkor, hogy mennyire nem érdekel senkit a *C* alapú *CGI* alkalmazások írása. Írásomban éppen ezért a *C* nyelv *CGI* programozásra való használatát fogom tárgyalni, valamint ismertetek néhány olyan helyzetet, amelyben alkalmazása számottevő előnyökkel jár.

Én három ok miatt használok *C*-t az alkalmazásaimban: gyors, nagy tudású és stabil. Bár a szájhagyomány másképp sugallja, saját méréseim alapján egyszerűbb feladatok végrehajtásakor a *C* és a *PHP* sebessége azonos, amikor pedig a feladat bonyolultabbá válik, a *C* lehengerlő győzelmet arat.

A *C* emellett rendkívül nagy tudású nyelv. Maga a nyelv ugyan csak egyfajta váznak mondható, ám az elérhető könyvtárak hihetetlenül széles választékával szinte mindent meg tudunk oldani, amire csak egy számítógép alkalmas lehet. Természetesen a *Perl* sem kezdő játékos ezen a területen, és egy pillanatig sem vitatom, hogy mindkét nyelv rendkívüli sokoldalúságot kínál használójának; bár szerintem a *C* könnyebben bővíthető.

Továbbá, a *C*-ben írt *CGI* programok üzembiztosabbak. Mivel a program teljes egészében lefordításra kerül, nem érzékeny az operációs rendszer által biztosított környezet megváltozásaira, ahogy például a *PHP*. Maga a nyelv is állandónak tekinthető, esetében nem kell olyan gyökeres változásokkal számolni, mint amilyenek a *PHP* használóinak idegeit borzolták az elmúlt években.

Az alkalmazás

Alkalmazásom egy egyszerű eseménylista, a jövőbeli események kezelésére, például üzleti megbeszélések időpontjának rögzítésére vagy akár egy templom eseményeinek ütemezésére használható. Rendelkezik egy – szándékom szerint – jelszóval védett felügyeleti felülettel, valamint egy nyilvános felülettel is, amely – kizárólag – a közeljövő eseményeit listázza ki. Az alkalmazás felületfüggetlen, beállításai futás közben módosíthatók.

1. kódrészlet MySQL séma

```
CREATE TABLE event (
  event_no int(11) NOT NULL auto_increment,
  event_begin date NOT NULL default '0000-00-00',
  name varchar(80) NOT NULL default '',
  location varchar(80) NOT NULL default '',
  begin_hour varchar(10) default NULL,
  end_hour varchar(10) default NULL,
  event_end date NOT NULL default '0000-00-00',
  PRIMARY KEY  (event_no),
  KEY event_date (event_begin)
)
```

Nem írtam saját adattárat, inkább adatbázist használok, a hozzá való csatlakozáshoz szükséges adatokat egy beállító fájlba helyezem el. A felület és a kód szétválasztását különálló fájlcsoporttal biztosítom.

A felügyeleti felület alkalmas az események listázására, szerkesztésére, mentésére és törlésére. Az események listázása az alapművelet, amennyiben más műveletet nem választunk ki. Az új és a meglévő eseményeket egyaránt lehet menteni. A felület tartalmaz egy rácsozatot, ebben jelenik meg az eseménylista, valamint egy részletes képernyőt, amin egy-egy esemény részletei is megtekinthetők.

Az alkalmazás adatbázissémája egyetlen táblából áll, ezt az *1. kódrészlettel* adjuk meg. Az alábbi séma ***MySQL***-hez készült, de természetesen bármilyen adatbázismotorhoz megírható egy ilyen kódrészlet.

A következő függvényekre mindenképpen szükség van a felügyeleti felület által biztosított szolgáltatások megvalósításához: `list_events()`, `show_event()`, `save_event()` és `delete_event()` (rendre események listázása, megjelenítése, mentése és törlése). Az adatbázis olvasását és írását is külön függvénycsoportba fogom elvonatkoztatni, így az egyes függvények egyszerűbbek maradnak, és egyszerűbb a hibakeresés is. Az adattároló felülethez a következő függvények szükségesek: `event_create()`, `event_destroy()`, `event_read()`, `event_write` és `event_delete` (esemény létrehozása, megsemmisítése, olvasása, írása és törlése). Mivel minden lehetőséget megragadok, hogy megkönnyítsem az

2. kódrészlet A beállítások futás közbeni megadását lehetővé tévő függvény

```
void config_read(char* filename, char** key,
                 char** value) {
  FILE* cfile;
  char tok[80];
  char line[2048];
  char* target;
  int i;
  int length;
  cfile = fopen(filename, "r");
  if (!cfile) {
    perror("config_read");
    return;
  }
  while(fgets(line, 2048, cfile)) {
    if ((target = strchr(line, '='))) {
      sscanf(line, "%80s", tok);
      for(i=0; key[i]; i++) {
        if (strcmp(key[i], tok) == 0) {
          target++;
          while(isspace(*target)) target++;
          length = strlen(target);
          value[i] = (char*)calloc(1, length + 1);
          strcpy(value[i], target);
          target = &value[i][length - 1];
          while(isspace(*target)) *target– = 0;
        }
      }
    }
  }
  fclose(cfile);
}
```

életemet, készítek egy `event_fetch_range()` (eseménytartomány lekérdezése) függvényt is, amivel adott tartományba eső eseményeket tudok kiválasztani. Erre legalább két helyen szükség lesz.
Következőként a rekordjaimat *C* adatszerkezeteknek, az adatbázis-lekérdezések eredményeit pedig láncolt listáknak kell megfeleltetnem. Az elvonatkoztatás lehetővé teszi, hogy – mivel a kódnak csak kis része foglalkozik közvetlenül az adattárolóval – viszonylag kis fáradság árán le tudjam cserélni az adatbázismotort, vagy meg tudjam változtatni az adatábrázolás módját.
A teljes forráskódot természetesen nyomtatásban nem jelentethetjük meg, ezért a ***Makefile*** társaságában a weboldalamról lehet letölteni (lásd az internetes forrásokat).

Eszközök

C használatakor az első leküzdendő akadály a szükséges eszközök beszerzése. Minimálisan egy *CGI*-feldolgozóra szükségünk lesz, ez fogja elérhetővé tenni számunkra a *CGI* adatait. Nagy valószínűséggel valamilyen adatbázis-kapcsolat is jól jön. Nem árt, ha a vezérlő logikát és a felületet valamilyen szinten szét tudjuk választani, így ugyanis az oldal átdolgozása nem jár a kód újabb és újabb újraírásával.
CGI feldolgozási célokra ***Thomas Boutell cgic*** könyvtárát ajánlom (lásd a forrásokat). Használata megdöbbentően

3. kódrészlet HTML sablonfüggvény

```
void html_get(char* path, char* file) {
  struct stat sb;
  FILE* html;
  char* buffer;
  char fullpath[1024];
  /* A fájl és az elérési út neve túllép a rendszer
  ➥korlátján */
  if (strlen(path) + strlen(file) > 1024) return;
  sprintf(fullpath, "%s/%s", path, file);
  if (stat(fullpath, &sb)) return;
  buffer = (char*)calloc(1, sb.st_size + 1);
  if (!buffer) return;
  html = fopen(fullpath, "r");
  fread((void*)buffer, 1, sb.st_size, html);
  fclose(html);
  puts(buffer);
  free(buffer);
}
```

egyszerű, és a CGI felület minden részéhez elérést biztosít. Ha inkább *C++* irányultságúak vagyunk, a ***cgicc*** könyvtárakat használhatjuk (lásd a forrásokat). Szerintem a ***Boutell*** könyvtárral könnyebb dolgozni.
A ***MySQL*** a ***UNIX*** alapú webes fejlesztéseknél gyakorlatilag szabványnak tekinthető, ezért ebben az esetben én is nála maradok. Minden komolyabb adatbázismotornak van *C* felületkönyvtára, vagyis mindenki olyan adatbázist választ, amilyet csak szeretne.
Jómagam elkészítem a saját felületfüggetlen eljárásaimat, de aki úgy gondolja, a ***libxml*** és a ***libxslt*** segítségével ugyanezt jóval kifinomultabb szinten oldhatja meg.

Futás közben megadott beállítások

Fontos tényező, hogy az adatbázis-kapcsolat beállításait futás közben is meg lehessen adni. A beállító függvény egy fájlnév és egy a beállító kulcsokat megadó karakterlánctömb alapján feltölt egy tömböt a megfelelő beállítási értékekkel. (Lásd a *2. kódrészletet*.) Ezután tetszőleges, az általam kiválasztott kulcsokkal tölthetek fel egy megfelelő karakterlánctömböt, az eredményt az értéktömbben kapom meg.

A felhasználói felület

A felhasználói felület két részből áll. Én, mint programozó, elsősorban a beviteli űrlapokkal és az ***URL***-karakterláncokkal foglalkozom. Mindenki más inkább arra figyel, hogy az űrlapomat körülvevő oldal hogyan néz ki, magát az űrlapot adottnak veszi. Ha mindkét fél elégedettségét biztosítani akarjuk, akkor külön kell választanunk az oldalt az űrlaptól és a programtól.
PHP és ***Perl*** alá számtalan sablonkönyvtár létezik, ám *C* alá nem nagyon találunk ***HTML*** sablonokat. A legjobb az, ha a *C* kód a kimenetnek csak a lehető legkisebb részét foglalja magába, a maradékot pedig ***HTML*** formátumú fájlokba helyezzük el, melyeket mindig a megfelelő időpontban jelenítünk meg. Erre alkalmas függvényt látunk a *3. kódrészletben*.
A kimenet kiadása előtt közölnünk kell a webkiszolgálóval és a böngészővel, hogy mit is szeretnénk átadni; a `cgiHeaderContentType()` pontosan ezt a célt szolgálja.

4. kódrészlet save_event(), CGI adatok feldolgozása

```
struct event* e;
e = event_create();
cgiFormInteger("eventno", &e->event_no, 0);
cgiFormStringNoNewlines("name", e->name, 80);
cgiFormStringNoNewlines("location",
                        e->location, 80);
/* Processing date fields */
cgiFormInteger("beginyear",
               &e->event_begin->year, 0);
cgiFormInteger("beginmonth",
               &e->event_begin->month, 0);
cgiFormInteger("beginday", &e->event_begin->day,
➥0);
cgiFormInteger("endyear", &e->event_end->year, 0);
cgiFormInteger("endmonth", &e->event_end->month,
➥0);
cgiFormInteger("endday", &e->event_end->day, 0);
/* Process begin & end times separately */
cgiFormStringNoNewlines("beginhour",
                        e->event_begin->hour, 10);
cgiFormStringNoNewlines("endhour",
                        e->event_end->hour, 10);
event_write(e);
cgiHeaderLocation(cgiScriptName);
```

A tartalom típusa (`content type`) `text/html`, ezért ezt használom átadott értékként. Minden oldal megjelenítése előtt az alábbi általános lépéseket kell követnünk:

```
cgiHeaderContentType("text/html");
html_get(path, pagetop.html);
A tartalom előállítása.
html_get(path, pagebottom.html);
```

Az űrlapok feldolgozása

Megoldottuk az oldalak és az űrlapok megjelenítését, most tehát gondoskodnunk kell az űrlap tartalmának feldolgozásáról. Numerikus és szöveges elemek beolvasására egyaránt szükség lehet, ezért a ***cgic*** könyvtár két függvényét veszem igénybe, ezek a `cgiFormStringNoNewlines()` és a `cgiFormInteger()`.
A ***cgic*** könyvtár tartalmazza a fő függvény megvalósítását, rám csak az int `cgiMain(void)` – túlnyomó részt ebben történik az űrlap feldolgozása – megírásának feladata hárul.
Ha a `show_event` függvénnyel egyetlen rekordot akarok megjeleníteni, akkor a ***CGI*** `eventno` átadott érték `event_no` (ez az elsődleges kulcs) értékét veszem.
A `cgiFormInteger()` függvény egy egész értéket kérdez le, és ha nincs megadva ***CGI*** átadott érték, akkor alapértelmezettet állít be.
A `save_event` szintén jó néhány adatot igényel az űrlapból. A dátumok bevitele fogas kérdés, ugyanis három adatrészből állnak: évből, hónapból és napból. Kezdő és záró dátumra egyaránt szükség van, vagyis összesen hat mező tartalmát kell feldolgozni. Szükség van még az esemény nevére, kezdő és záró időpontjára (ezek karakterláncok, ugyanis önmagukban is lehetnek események, mint például napkelte és napnyugta) és helyére. A *4. kódrészlet* mindennek a megvalósítását szemlélteti.

5. kódrészlet A küldés gombok kezelése

```
char* command[5] = {"List", "Show",
                    "Save", "Delete", 0};
void (*action)(void)[5] = {list_events,
  show_event, save_event, delete_event, 0};
int result;
cgiFormSelectSingle("do", command, 4, &result, 0);
action[result]();
```

A 4. kódrészletben a `cgiHeaderLocation()` is szerepel, ennek feladata a felhasználó új oldalra irányítása. Miután elmentettem az elküldött adatokat, mindig az eseménylista oldalt szeretném megjeleníteni. Meghatározott karakterlánc helyett a ***libcgic*** által biztosított változók egyikét, a `cgiScriptName`-et használom, ezzel azt érem el, hogy a program neve működésének megzavarása nélkül megváltoztatható.
Végül kell valamilyen megoldás az adatok elküldésére szolgáló (***submit***) gombok kezelésére. Ezek a legösszetettebb beviteli megoldások, az elindítandó függvény ugyanis esetükben az értéküktől függ, illetve szükség esetén alapértelmezett értéket is kell választani. A ***cgic*** könyvtárnak van egy függvénye, a `cgiFormSelectSingle()`, amely pontosan ezt végzi el. Működéséhez a lehetséges értékeknek egy karakterlánctömbben kell szerepelniük. Egy egész értékbe helyezi a megfelelő átadott érték tömbbeli indexét; ha pedig nem talál egyezést, akkor az alapértéket használja.
A függvénymutatókról bővebb tájékoztatás a források között található. Aki nem békült még meg a függvénymutatókkal, az `switch` utasítással is megoldhatja mindezt. Jómagam inkább a függvénymutatókat szeretem, ez a megoldás ugyanis tömörebb; régebbi kódjaimban ugyanakkor természetesen a `switch` utasítást is használtam.

Az adatbázisrendszer

A ***MySQL*** kezelése ***C*** alól nagyon hasonló a ***PHP*** alóli kezeléshez – már amennyiben ez mond valamit. A karakterláncokban szereplő problémás karakterek, például a kettős idézőjelek és a visszaperjelek eltávolítására a ***MySQL*** megfelelő függvényeit kell használnunk, de más eltérést nem nagyon fogunk találni. A show_event() függvény esetében az elsődleges kulcs alapján kell lekérdeznünk egy rekordot. A hibakezelés miatt a kód meghízik ugyan, de három alaputasítás adja a lényeget. A `mysql_query()` hívásakor lefut a ***MySQL*** utasítás, aminek keletkezik valamilyen eredménye. Ezután a `mysql_store_result()` lekérdezi az eredménykészletet a kiszolgálótól. Végül a `mysql_fetch_row()` egyetlen `MYSQL_ROW` változót emel ki az eredménykészletből.
A `MYSQL_ROW` változót karakterláncok tömbjeként (`char**`) kezelhetjük. Ha az adatok valamelyike numerikus, és numerikusként is szeretnénk használni, akkor át kell alakítanunk. Jelenlegi programunkban például a dátumot három különálló számérték formájában érdemes kezelni. Mivel a kapott adat ***ÉÉÉÉ-HH-NN*** szerkezetű, a `sscanf()` függvénnyel választhatjuk szét az összetevőit. (*6. kódrészlet*)
Az adatok adatbázisba írása érdekesebb, itt ugyanis ügyelni kell a problémás karakterek kiszűrésére.
Ezt a *7. kódrészlet* szemlélteti.

6. kódrészlet Adatok lekérdezése MySQL-ből

```
MYSQL_RES* res;
MYSQL_ROW row;
int beginyear;
int beginmonth;
int beginday;
if (mysql_query(db, sql)) {
  print_error(mysql_error(db));
  return;
}
if((res = mysql_store_result(db)) == 0) {
  print_error(mysql_error(db));
  return;
}
if ((row = mysql_fetch_row(res)) == 0) {
  print_error("Nincs ilyen számú esemény.");
  return;
}
sscanf(row[0], "%d-%d-%d", &beginyear, &beginmonth,
       &beginday);
```

Az `escapedname` ugyanazt a karakterláncot tartalmazza, mint a `name`, de a *MySQL* különleges karakterei nélkül; így már nyugodtan beilleszthetjük egy *SQL*-utasításba. Fontos, hogy a felhasználók által megadott karakterláncokat mindig tisztítsuk meg, ellenkező esetben rosszindulatú személyek kihasználhatják hanyagságunkat, és kellemetlen dolgokat művelhetnek az adatbázisunkkal.

CGI programok hibakeresése

A *C* programok hibáinak felderítése elég kellemetlen tevékenység, ugyanis általában szegmentációs hibát okoznak, és semmilyen jelzést nem kapunk a hiba forrásáról. A hibakereső programok más alkalmazások esetében kiválóan megfelelnek, ám a *CGI* programoknál a bemenő adatok fogadásának módja miatt más a helyzet.
Ilyenkor siet segítségünkre a *cgic* könyvtárban található `capture` nevű *CGI* program, mely a neki küldött *CGI*-bemeneteket egy fájlba rögzíti. A fájl nevét a `capture` forráskódjában adhatjuk meg. Ha *CGI* programunkon hibakeresést kell végeznünk, akkor `cgiMain()` függvényének elejére adjunk hozzá egy `cgiReadEnvironment(char*)` hívást. Ügyeljünk arra, hogy a `filename` átadott érték megegyezzen a rögzítéshez megadottal. Ezután a problémás adatot a kérésünkben szereplő űrlap vagy parancsfájl alapértelmezett műveletévé téve küldhetjük el rögzítésre. Végül a *GDB*-vel vagy kedvenc hibakeresőnkkel bogozhatjuk ki, hogy kódunk pontosan milyen hibát okozott.
A későbbi hibakeresést és fejlesztést bizonyos lépésekkel leegyszerűsíthetjük. Bár ezek minden program fejlesztésére érvényesek, *CGI* programozásnál különösen kifizetődők. Ne feledjük, egy-egy függvénynek egyetlen, és csakis egyetlen feladatot adjunk, továbbá tesztelését minél hamarabb kezdjük el, és a lehető leggyakrabban ismételjük meg.
A kód írása közben érdemes minden egyes függvényt azonnal kipróbálni, és ellenőrizni, hogy valóban a tőle elvárt módon működik-e. Azt is érdemes megvizsgálni, hogy hibás adatok megadására hogyan válaszol, ugyanis nagyon valószínű, hogy élete során a függvény előbb-utóbb valóban fog hibás adatokat kapni. Ha minderre jó előre gondolunk, akkor sok kellemetlenséget spórolhatunk meg magunknak.

7. kódrészlet A felhasználó által megadott adatok kezelése MySQL-lel

```
char name[11];
char escapedname[21];
cgiFormStringNoNewlines("name", name, 10);
mysql_real_escape_string(db, escapedname, name,
                         strlen(name));
```

Telepítés

A fejlesztésre és a futtatásra használt számítógép az esetek nagy részében nem ugyanaz. A fejlesztői rendszert ezért próbáljuk a lehető legnagyobb mértékben azonossá tenni a termelési rendszerrel. Én például a programjaimat általában *Linux* vagy *OpenBSD* alatt fejlesztem, futtatásuk viszont szinte mindig *FreeBSD* alatt történik.
Amikor felkészülünk a termelési rendszer felépítésére vagy telepítésére, akkor különösen fontos, hogy tisztában legyünk a könyvtárváltozatok közötti különbségekkel. Azt, hogy kódunk mely dinamikus könyvtárakat használja, az `ldd` paranccsal tekinthetjük meg. Érdemes ellenőrizni, mert meglepetéseket okozhatnak a könyvtárak miatt felmerülő további függőségek.
Ha a könyvtárváltozatok közel állnak egymáshoz, ami általában a fő változatszámok megegyezésében nyilvánul meg, akkor különösebb gondokra nem kell számítanunk. A külső üzemeltetésű weboldalra telepített programoknál gyakori a fejlesztői és a termelési rendszer által tartalmazott változatok eltérése.
Ilyenkor én saját, helyi változatot szoktam fordítani a könyvtárból. Eltávolítom a könyvtár megosztott változatát, és a rendszer által tartalmazott helyett a helyi változatot építem be. Természetesen ilyenkor a futtatható fájl megnő, ám a fennhatóságunkon kívüli könyvtáraktól való függése is megszűnik.
Miután lefordítottuk a futtatható fájlt a termelési rendszeren, az `ldd` ismételt kiadásával ellenőrizzük, hogy az összes dinamikus könyvtár megtalálható-e. A könyvtárak helyi változatainak beépítésekor különösen gyakran fordul elő, hogy elfeledkezünk a dinamikus változat eltávolításáról, amit persze futás közben nem fog találni a program (és az `ldd`). A fordítást sose kapkodjuk el, újra és újra fordítsunk és ellenőrizzünk, míg végül el nem tűnnek a könyvtárak megtalálásával kapcsolatos hibák.

Sebesség: CGI kontra PHP

Általános nézet, hogy a *CGI* felületet használó programok lassabbak, mint a valamilyen kiszolgáló modul – mint a *mod_php* és a *mod_perl* – által biztosított nyelvet használók. Mivel a webes alkalmazások fejlesztését *PHP*-ben kezdtem, ezt fogom a *C*-ben írt *CGI*-k sebességének vizsgálatakor kiindulási alapként venni. Nyilván mindebből a *C* és a *Perl* közötti sebességkülönbségekre nézve nem következik semmi.
Az összehasonlításnál a külső adatbázis-felületet vettem (`events.cgi` és `events.php`), ugyanis mindkettő azonos

eljárást alkalmaz a felület elkülönítésére. A belső felületet nem teszteltem, ugyanis a külső hívások mellett a belsők szerepe szinte elenyésző.

Az ***Apache Benchmark*** segítségével mindkét változatot tízezer lekérdezéssel terheltem le, amilyen gyorsan a kiszolgáló képes volt azokat kezelni. A ***C*** alapú változat átlagos végrehajtási ideje 581 ms, a ***PHP*** alapú változaté pedig 601 ms volt. Mivel a két érték közel esett egymáshoz, feltételeztem, hogy a méréseket megismételve némi szórást észlelek majd. Így is volt, de végeredményként elmondhatom, hogy a ***C*** alapú változat az esetek nagyobb hányadában volt gyorsabb a ***PHP*** alapúnál.

Munkáim során egy összetettebb felületelválasztó könyvtárat használok, ez a libtemplate (lásd a forrásokat). A könyvtárból ***PHP*** és ***C*** változat egyaránt létezik. Amikor az eseményütemező változatait a ***libtemplate*** segítségével hasonlítottam össze, akkor a ***C*** alapú változat határozottan jobb válaszidőkkel futott. A ***C*** alapú változat átlagos futási ideje 625 ms volt; nem sokkal több, mint az egyszerűbb változaté.

A ***PHP*** alapú változat átlagos futási idejére 1957 ms-ot kaptam. Azt is érdemes megjegyezni, hogy a terhelés a ***PHP*** alapú változat futása közben a ***C*** alapú változat futásakor mértnek nagyjából kétszerese volt. A tesztek végrehajtása közben más komolyabb alkalmazás nem futott a gépen, és más felhasználók sem terhelték.

A két ***C*** alapú változat egymáshoz közeli futtatási ideje arra utal, hogy az idő nagy része a program betöltésével telik. Ha ez megtörtént, a futtatás már gyorsan megy. A ***PHP*** ezzel szemben viszonylag lassan fut. Természetesen a ***PHP*** esetében is be kell tölteni a programot a memóriába, ám itt a fordítást – amelyen a ***C*** alapú program már túlesett – is el kell végezni.

Összefoglalás

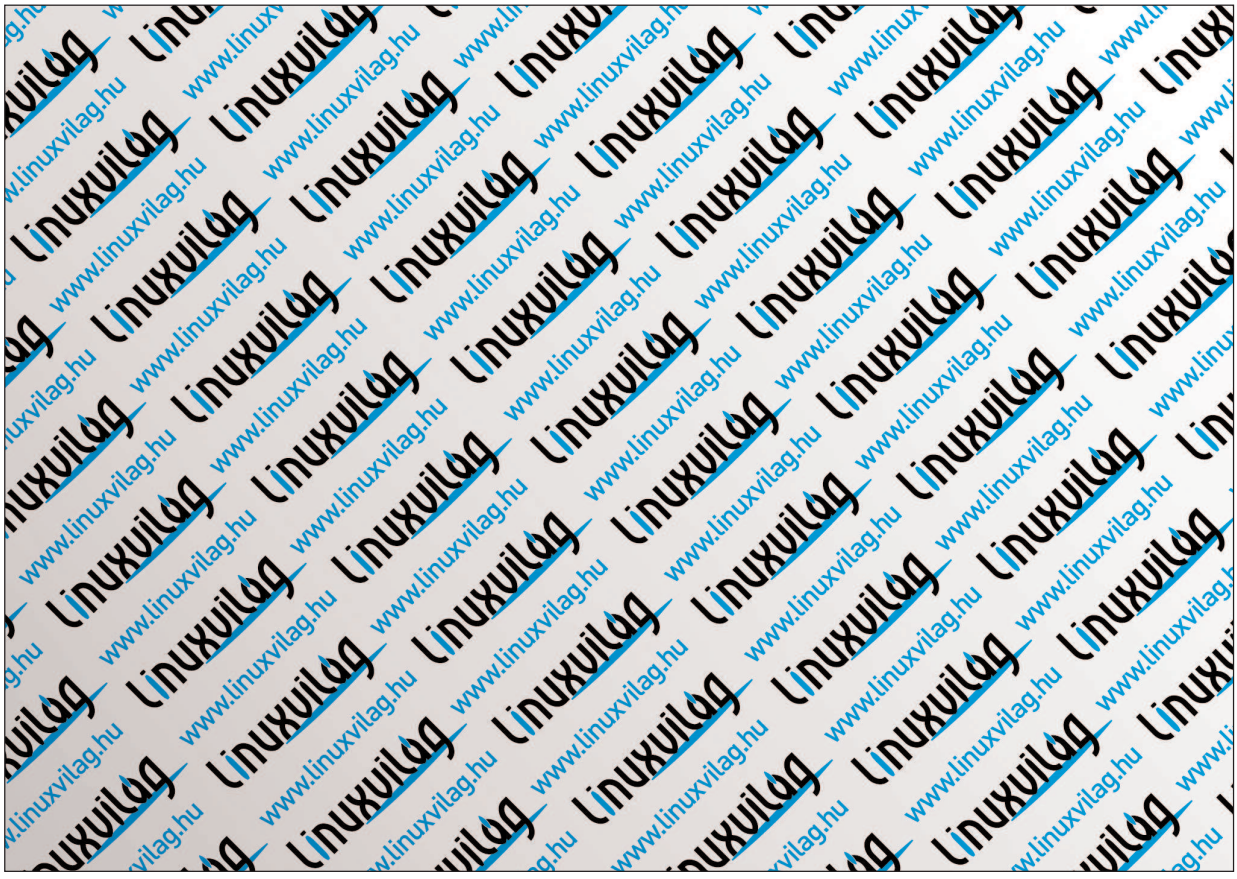
Megfelelő eszközökkel és némi tapasztalattal a ***C*** alapú ***CGI*** alkalmazások fejlesztése nem nehezebb, mint a ***Perl*** vagy ***PHP*** alapúaké. Jómagam mindkettővel rendelkezem, és egyértelműen a ***C***-t választottam a ***CGI***-k fejlesztésére.

A ***C*** előnyei főként bonyolultabb műveletek elvégzésekor és a hosszú távú stabilitás terén mutatkoznak meg. A ***PHP***-val szemben teljesen érzéketlen a kiszolgálót érintő változásokra. Hacsak nem töröljük a gépről valamelyik megosztott könyvtárat, például a ***libc***-t vagy ***libmysqlclientet***, a ***C*** alapú változatot rendkívül nehéz tönkretenni. A ***C*** alapú programok futásuk gyorsasága miatt összetettebb adatfeldolgozó műveletek végrehajtásakor egyértelműen jobb választásnak bizonyulnak.

Linux Journal 2005. április, 132. szám

A cikk forrásai: ➲ ***www.linuxjournal.com/article/8058***

Clay Dowling a Lazarus Internet Development (www.lazarusid.com) elnöke. A programozás mellett hobbija a sörfőzés és a borkészítés. A clay@lazarusid.com címen érhető el.



Squid alapú forgalomszabályozó és felügyeleti rendszer

Amikor a hálózatunkon folyó forgalom túlnyomó része webes forgalom lett, egyetemünk a nyílt forrású Squid gyorsítótárazó programra építve üzembe helyezett egy a hozzáférések követésére és korlátozására alkalmas rendszert.

Bármilyen szervezet számítógépes hálózatáról is legyen szó, az internetelérés az egyik legfontosabb és leginkább igényelt szolgáltatás. ***Olifer*** és ***Olifer*** a ***Computer Networks: Principles, Technologies and Protocols*** című könyvükben azt írják, hogy az elmúlt 10-15 évben a 80/20-as arány a belső és a kimenő forgalom között megfordult, és jelenleg a forgalom 80 százaléka kifelé irányul (lásd az internetes forrásokat). A hozzáférés sebessége, a szolgáltatások száma és a rendelkezésünkre álló tartalom mennyisége folyamatosan növekedik. Az internetelériéshez fűződő felhasználói hozzáférés-vezérlés kérdése ezzel párhuzamosan egyre fontosabbá válik. Maga a probléma természetesen nem új, ám bizonyos vonatkozásai az idők során megváltoztak. Írásunkban az elérhető korszerű megoldásokat tekintjük át, példaként a ***Bashkir State Pedagogical University (Baskíri Állami Pedagógiai Egyetem, BSPU)*** hálózatát véve.
Elsőként határozzuk meg az internet elérését szabályozó és felügyelő rendszerrel szemben támasztott elvárásokat:

- Felhasználói fiókok támogatása és kezelése
- Felhasználói forgalom számlázása és szabályozása
- Háromféle mód a felhasználói forgalom szabályozására: havi, heti és napi
- A mozgó felhasználók támogatása (ők az egyes alkalmakkor különböző számítógépeket használnak az internet elérésére; ilyenek például a hallgatók)
- Napi és heti kimutatások a rendszer állapotáról, jelentések a webes és elektronikus levélforgalomról
- Web alapú kimutatások és rendszerfelügyelet

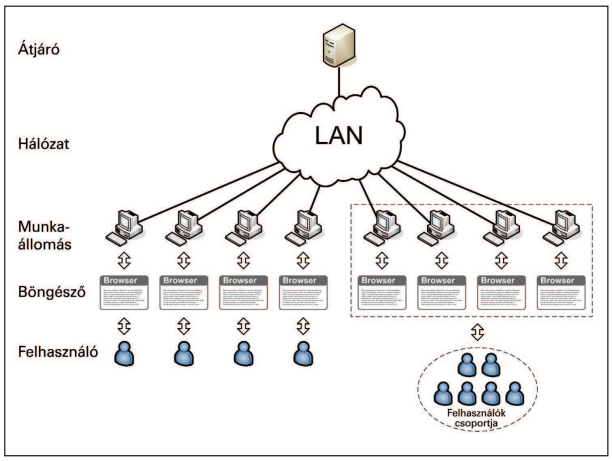
Láthatóan mindezek az elvárások semmilyen módon nem hatnak ki a rendszer megvalósításának fázisára, vagyis képzeletünknek sem jelentenek korlátokat. A problémát tehát általános jelleggel szemléltük, majd kidolgoztuk a megoldását. Írásunk további részeiben a végső döntésünk felé vezető ötleteket és megfontolásokat ismertetjük.

A kérdéskör általános elemzése

A legnépszerűbb szolgáltatás, a ***World Wide Web (WWW)*** példáján röviden tekintsük át, hogy az internetelérés folyamata milyen lépésekből áll:

1. A felhasználó elindítja a böngészőprogramot, majd beírja a kívánt ***URL***-t.
2. A böngésző vagy közvetlen kapcsolatot létesít a ***WWW***-kiszolgálóval az átjárón keresztül, utóbbi akár címfordítást vagy más csomagmanipulációs műveletet is végezhet, vagy proxykiszolgálón keresztül csatlakozik, mely gondosan elemzi az ügyfél kérését és ellenőrzi, hogy a kért adatok szerepelnek-e a gyorsítótárában. Ha nem rendelkezik a kért adatokkal, esetleg azok elavultak, akkor a proxykiszolgáló a saját nevében kapcsolódik a ***WWW***-kiszolgálóhoz.
3. A kapott adatokat megkapja az ügyfél.
4. A böngészőprogram lezárja vagy ébrenléti állapotba állítja a kapcsolatot.

Az *1. ábrán* az internetelérés folyamatát szemléltettük. A folyamat fő résztvevői a felhasználó, az ügyfélprogram (böngésző és operációs rendszer), a munkaállomás vagy egyéb ügyfélhardver, a hálózati eszközök és az átjáró vagy proxykiszolgáló. A hálózatra további, például felhasználóhitelesítést végző kiszolgálók is csatlakozhatnak, például ***Microsoft Windows*** tartományvezérlők, ***OpenLDAP***- vagy ***NIS***-kiszolgálók.



1. ábra A felhasználók internetelérésének folyamata

Mint az 1. ábra is szemlélteti, a felhasználók és a munkaállomások között egy-egy és több-több megfeleltetés egyaránt lehet. Az egyetem munkatársai például a legtöbb esetben saját számítógéppel rendelkeznek.
Elsődleges céljaink a felhasználói forgalom számlázása, a felhasználók hitelesítése, a felhasználók hozzáférésének ellenőrzése, a felügyelet és jelentések készítése.
Ezek a célok egymástól nagymértékben függetlenek, és mindegyiket többféle módon is el lehet érni. A hitelesítés, a forgalomszámlázás és a hozzáférés-vezérlés feladata a fenti séma elemeinek bármelyikéhez hozzárendelhető, a legjobb megoldás azonban kétségtelenül az lenne, ha az összes feladat elvégzése egyetlen helyen összpontosulna.
A hozzáférés-vezérlés ügyfél- és kiszolgálóoldalon egyaránt megvalósítható. Az ügyféloldali hozzáférésvezérlés különleges, a felhasználók hitelesítésére is képes ügyfélprogram használatát teszi szükségessé.
A hozzáférés-vezérlés kiszolgálóoldali megvalósításának kétféle módja van: tűzfal vagy proxykiszolgáló alkalmazása. A tűzfal alapú megoldás nem alkalmas a felhasználók hitelesítésére, hiszen a hálózati csomagok csak *IP*-címeket tartalmaznak, amelyek semmilyen módon nem kötődnek a felhasználónevekhez. Ha tűzfalat akarunk használni, akkor erre kétféle megoldás kínálkozik: saját felhasználóhitelesítést alkalmazó *VPN* használata vagy dinamikus felhasználó/IP-cím hozzárendelések megadása. Ezeket a megoldásokat külső eszközökkel valósíthatjuk meg.
A legegyszerűbb megoldás mégis egy proxykiszolgáló használata, mely képes a böngészőn keresztül végzett felhasználóhitelesítésre. A böngésző alapú hitelesítésnek háromféle módszere létezik:

- Alapszintű hitelesítés – Egyszerű és széles körben alkalmazott megoldás, melyet az internetes böngészők és proxykiszolgálók túlnyomó része támogat. Fő hátránya, hogy a felhasználó jelszava a hálózaton keresztül titkosítatlanul kerül továbbításra.

- A kivonat alapú hitelesítés egy megbízhatóbb megoldás, hátránya a szoftveres támogatás hiánya.

- Az *NTLM* alapú hitelesítés csak a *Microsoft* termékekre épülő hálózati rendszerekben alkalmazható. A windowsos munkaállomásokat tartalmazó hálózatokban ugyanakkor nemcsak elfogadható, de kifejezetten előnyös, márpedig tudomásunk szerint *Oroszországban* ezek a rendszerek terjedtek el leginkább. A fő előny itt az, hogy lehetőség nyílik a proxyhitelesítés összevonására a windowsos és a *Samba* alapú tartományvezérlőkre szóló hitelesítéssel.

A körülmények elemzésével és néhány ötlet nyomán kétféle rendszert dolgoztunk ki:

1. *VPN*, *PPTP* használatával, a tűzfal beépített szolgáltatásaira építve. A *VPN*-kiszolgáló hagyományosan *FreeBSD* volt, ezért az *ipfw* tűzfalfelületet és az *mpd PPTP*-kiszolgálót alkalmaztuk. A forgalomszabályozást az ingyenesen elérhető *NetAMS* rendszer végezte.

2. *Squid*-alapú hozzáférés-vezérlési és felügyeleti rendszer.

Az első rendszert *Vladimir Kozlov* fejlesztette ki, feladata az egyetem az internet elérésére állandó gépet használó munkatársai által használt kapcsolatok biztosítása. Legnagyobb hátránya az, hogy ügyféloldali *VPN*-telepítést igényel. Ez elég komoly gondot jelent, ha a hálózat elemei szétszórtan helyezkednek el, és a felhasználók kevésbé jártasak a számítógép használatában.
A második megoldás *Tagir Bakirovhoz* kötődik, segítségével történik az egyetemi felhasználók túlnyomó részének csatlakoztatása – ők azok, akik nem rendelkeznek állandó géppel. Ennek kapcsán a fő problémát a fejlesztés bonyolultsága jelentette; a továbbiakban ennek részleteit tárgyaljuk.

Squid-alapú internetes hozzáférés-vezérlési és felügyeleti rendszer

Mindenek előtt szeretnénk felhívni rá a figyelmet, hogy az itt szereplő elérési utak mindig a *Squid* forrásának alapkönyvtárához viszonyítva értendők, ami a mi esetünkben a */usr/local/src/squid-2.5STABLE7/* volt. A *Squid* beszerzésével, lefordításával és használatával kapcsolatosan a *Squid* webhelyén lehet részletes útmutatást találni. Fussuk át a *Squid* néhány jellemzőjét, ezek a *Squid Programming Guide*-ból származnak.
A *Squid* egy egyfolyamatos proxykiszolgáló. Minden ügyfél *HTTP*-kéréseit a főfolyamat kezeli. Futása lényegében visszahívó függvények sorozata. A visszahívó függvény végrehajtására akkor kerül sor, amikor a be/kivitel elvégezhető vagy valamilyen más esemény történt. Amikor a visszahívó függvény végzett, bejegyzi a következő be/kiviteli művelet visszahívó függvényét.
A *Squid* alapját a *select(2)* és a *poll(2)* rendszerhívás adja, melyek be/kiviteli eseményekre várakoznak fájlleírók egy csoportján. A *Squid* ezekkel végzi el a megnyitott fájlleírók be/kiviteli műveleteit. A `select()` rendszerhívást a `comm_select()` függvény adja ki, mely a teljes `fd_table[]` tömböt letapogatja, kezelő-

függvényeket keresve. Az eljárás minden kész leíróhoz meghívja a megfelelő kezelőt. A kezelőfüggvények bejegyzése a `commSetSelect()`, a lezáró kezelők meghívása pedig a `comm_close()` függvénnyel történik. A lezáró kezelők feladata a fájlleírókhoz rendelt adatszerkezetek felszabadítása. Ez az oka annak, hogy minden hívássorozatot a `comm_close()` meghívásának kell zárnia.
Érdekes ***Squid***-szolgáltatás az ügyfél/IP-cím adatbázis támogatása. Kódja a ***src/client_db.c*** fájlban található. Alapeleme egy kivonat alapján indexelt tábla (`client_table`), mely `ClientInfo` adatszerkezetekre vezető mutatókat tartalmaz. Ezek az adatszerkezetek különböző adatokat tartalmaznak a ***HTTP***-ügyfélről és az ***ICCP*** proxykapcsolatokról (mint például kérés-, forgalom- és időszámlálók). A ***src/structs.h*** fájl vonatkozó kódrészlete:

```
struct _ClientInfo {
    /* elsőnek kell lennie */
    hash_link hash;
    struct in_addr addr;
    struct {
        int result_hist[LOG_TYPE_MAX];
        int n_requests;
        kb_t kbytes_in;
        kb_t kbytes_out;
        kb_t hit_kbytes_out;
    } Http, Icp;
    struct {
        time_t time;
        int n_req;
        int n_denied;
    } cutoff;
    /* a jelenleg létrejött kapcsolatok
    ➥száma */
    int n_established;
    time_t last_seen;
};
```

Néhány fontos átfogó és helyi függvény az ügyféltábla kezeléséhez:

- `clientdbInit()` – Átfogó, az ügyféltábla kezdeti értékadását végző függvény.
- `clientdbUpdate()` – Átfogó függvény, szükség szerint a táblában lévő rekordot frissíti vagy új rekordot hoz létre.
- `clientdbFreeMemory()` – Átfogó függvény, törli a táblát és felszabadítja a hozzárendelt memóriát.
- `clientdbAdd()` – Helyi függvény, a `clientdbUpdate()` hívja meg; hozzáadja a rekordot a táblához, illetve ütemezi a szemétrekordokat összegyűjtő eljárást.
- `clientdbFreeItem()` – Helyi függvény, a `clientdbFreeMemory()` függvény hívja meg, és egyetlen rekordot távolít el a táblából.
- `clientdbSheduledGC()`, `clientdbGC()` és `clientdbStartGC()` – Helyi, a szemétrekordokat összegyűjtő eljárást megvalósító függvények.

Párhuzamba állítva a rendszerrel szemben támasztott követelményeket és a meglévő ügyféladatbázis által nyújtott lehetőségeket, kijelenthetjük, hogy az alapszolgáltatások egy része máris megvalósult, kivéve az ügyfelek felhasználóneve szerinti indexelést.
A meglévő ügyfélstatisztika-adatbázis további súlyos hiányossága, hogy az adatok frissítése csak azt követően történik meg, hogy az ügyfél a teljes kért tartalmat megkapta.
Munkánk során egy másik párhuzamos és független, ügyfél/felhasználó adatbázist készítettünk, kisebb módosításokkal a ***src/client_db.c*** fájlban található kódot felhasználva. A felhasználói statisztikák a `ClientInfo_sb` adatszerkezetbe kerülnek. Az ***src/structs.h*** fájl vonatkozó része:

```
#ifdef SB_INCLUDE
#define SB_CLIENT_NAME_MAX_LENGTH 16
struct _ClientInfo_sb {
    /* elsőnek kell lennie */
    hash_link hash;
    char *name;
    unsigned int GID;
    struct {
        long value;
        char type;
        long cur;
        time_t lu;
    } lmt;
    /* HTTP-kérések számlálója */
    int Counter;
};
#endif
```

Az ügyféladatbázis kezelését a következő – az előbbiekhez roppant hasonló – átfogó és helyi függvények végzik:

- `clientdbInit_sb()` – Átfogó, az ügyféltábla kezdeti értékadását végző függvény.
- `clientdbUpdate_sb()` – Átfogó függvény, mely frissíti a táblában lévő rekordot, a korlát elérésekor leválasztja az ügyfelet, illetve szükség esetén a `clientdbAdd_sb()` meghívásával hozzáadja az új ügyfelet.
- `clientdbEstablished_sb()` – Átfogó függvény, mely megszámlálja az ügyfélkéréseket, és rendszeres időközönként beírja a megfelelő rekordot a fájlba, a korlát elérésekor leválasztja az ügyfelet, illetve szükség esetén a `clientdbAdd_sb()` függvény meghívásával hozzáadja az új rekordot.
- `clientdbFreeMemory_sb()` – Átfogó függvény, törli a táblát és felszabadítja a hozzárendelt memóriát.

1. kódrészlet A clientWriteComplete() és a clientReadRequest() függvény egyes részei az src/client_side.c fájlból

```
static void
clientWriteComplete(int fd,
                    char *bufnotused,
                    size_t size,
                    int errflag,
                    void *data)
{
    clientHttpRequest *http = data;
...
    if (size > 0)
    {

kb_incr(&statCounter.client_http.kbytes_out,
                size);
/*-Itt kezdődik az SB rész---------------------*/
#ifdef SB_INCLUDE
    if (http->request->auth_user_request)
    {
        if ( authenticateUserRequestUsername(
                http->request-
>auth_user_request) )
            if (!clientdbUpdate_sb(
      authenticateUserRequestUsername(
          http->request->auth_user_request),
          size) )
            {
                comm_close(fd);
                return;
            }
    }
#endif
/*---------------------------------------------*/
        if (isTcpHit(http->log_type))
            kb_incr(
      &statCounter.client_http.hit_kbytes_out,
      size);
    }
...
}
...
static void
clientReadRequest(int fd, void *data)
{
    ConnStateData *conn = data;
    int parser_return_code = 0;
    request_t *request = NULL;
    int size;
    void *p;
    method_t method;
    clientHttpRequest *http = NULL;
    clientHttpRequest **H = NULL;
    char *prefix = NULL;
    ErrorState *err = NULL;
    fde *F = &fd_table[fd];
    int len = conn->in.size - conn->in.offset - 1;
...
    /* A kérés törzsének feldolgozása, ha van
    ➥ilyen */
    if (conn->in.offset > 0 &&
        conn->body.callback != NULL)
    {
        clientProcessBody(conn);
    }
    /* A következő kérés feldolgozása */
    while (conn->in.offset > 0 &&
           conn->body.size_left == 0)
    {
        int nrequests;
        size_t req_line_sz;
...
        /* Kérés feldolgozása */
        http = parseHttpRequest(conn,
            &method,
            &parser_return_code,
            &prefix,
            &req_line_sz);
        if (!http)
            safe_free(prefix);
        if (http) {
...
            if (request->method ==
            ➥METHOD_CONNECT)
            {
                /* Kérések beolvasásának
                ➥leállítása... */
                commSetSelect(fd,

COMM_SELECT_READ,
                                    NULL,
                                    NULL,
                                    0);
                clientAccessCheck(http);
/*-Itt kezdődik az SB rész---------------------*/
#ifdef SB_INCLUDE
                if(http->request
                ➥->auth_user_request)
                {
                    if (
      authenticateUserRequestUsername(
          http->request->auth_user_request
      )!=NULL)
                    {
                        if(!clientdbCount_sb(
          authenticateUserRequestUsername(
              http->request
              ➥->auth_user_request)))
```

1. kódrészlet (folytatás)

```
                    {
                        comm_close(fd);
                        return;
                    }
                }
            }
#endif
/*-------------------------------------------*/
                break;
            } else {
                clientAccessCheck(http);
/*-Itt kezdődik az SB rész--------------------*/
#ifdef SB_INCLUDE
                if(http->request
                ➥->auth_user_request)
                {
                    if (
      authenticateUserRequestUsername(
           http->request->auth_user_request
      )!=NULL)
                    {
                        if(!clientdbCount_sb(
      authenticateUserRequestUsername(
           http->request
           ➥->auth_user_request)))
                    {
                        comm_close(fd);
                        return;
                    }
                }
            }
#endif
/*-------------------------------------------*/
/* while offset > 0 && body.size_left
➥== 0 */
                continue;
            }
        } else if (parser_return_code == 0) {
...
/* while offset > 0 && conn->body.size_left
➥== 0 */
    }
...
}
```

- `clientdbAdd_sb()` – Helyi függvény, a `clientdbUpdate_sb()` hívja meg; hozzáadja a rekordot a táblához, illetve ütemezi a szemétrekordokat összegyűjtő eljárást.

- `clientdbFlushItem_sb()` – Helyi függvény, a `clientdbEstablished_sb()` és a `clientdbFreeItem_sb()` hívja meg; a megadott rekordot írja ki a fájlba.

- `clientdbFreeItem_sb()` – Helyi függvény, a `clientdbFreeMemory_sb()` függvény hívja meg, és egyetlen rekordot távolít el a táblából.

- `clientdbSheduledGC_sb()`, `clientdbGC_sb()` és `clientdbStartGC_sb()` – Helyi függvények, a szemétrekordokat összegyűjtő eljárást valósítják meg.

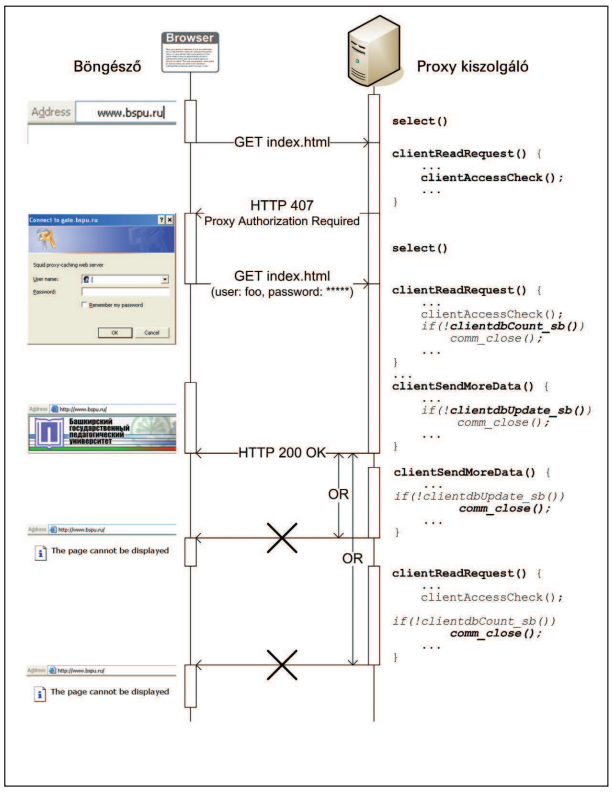
Az ügyféladatbázis kezdeti értékadásának és elengedésének megvalósítása az eredeti táblához hasonlóan az ***src/main.c*** fájlban található. Kódunk legfontosabb jellegzetessége a `clientdbUpdate_sb()` és a `clientdbEstablished_sb()` függvény meghívása a ***src/client_side.c*** fájl ügyféloldali eljárásaiban:

- A `clientdbUpdate_sb()` függvény meghívása a külső `clientWriteComplete()` függvényből, mely az adatok ügyfél felé történő elküldéséért felelős.

- A `clientdbEstablished_sb()` függvény meghívása az ügyfél kérését feldolgozó `clientReadRequest()` függvényből.

Az *1. kódrészlet* az ***src/client_side.c*** fájlból a `clientWriteComplete()` és a `clientReadRequest()` függvény megfelelő részeit tartalmazza.
A kód működése rendkívül egyszerű. A *2. ábra* az ügyfelektől érkező kérések feldolgozásának folyamatát szemlélteti a rendszer szemszögéből. Minden az ügyfelektől érkező kérés tartalmazza a hitelesítési adatokat, köztük a felhasználónevet is. A `clientdbUpdate_sb()` függvény megkeresi a kérésben szereplő felhasználónévhez tartozó `ClientInfo_sb` rekordot. Ha nincs ilyen rekord, akkor a hitelesítési fájlok tartalma alapján új `ClientInfo_sb` rekordot hoz létre. Ha a felhasználók túllépik korlátjukat, akkor a `comm_close()` függvény azonnal megszakítja a kapcsolatukat.
A `clientdbEstablished_sb()` függvény meghívása az ügyfélkérések számának szabályozását szolgálja, illetve az aktuális felhasználói információk elmentését a hitelesítési fájlokba, minden `SB_MAX_COUNT` kérés után. A hitelesítési fájlok neve – a unixos világban megszokott módon – `passwd` és `group`. A `passwd` fájl tartalmazza a felhasználói, a `group` pedig a csoportokkal kapcsolatos adatokat. Például:

```
`passwd':
#<név>:<teljes név>:<csoportazonosító>:
#<jelenlegi korlát>:<korlát utolsó frissítésének
➥időpontja>
```



2. ábra Ügyféltől beérkezett kérés feldolgozásának folyamata

```
tagir:Tagir Bakirov:1:6567561:12346237467
`group':
#<név>:<teljes név>:<csoportazonosító>:
#<csoport korlátja>:<csoportkorlát típusa>
users:BSPU users:1:10000000:D
```

Háromféle korlát létezik: `D` (***daily, napi***), `W` (***weekly, heti***) és `M` (***monthly, havi***). A `passwd` és a `group` fájl nevét és elérési útját a *Squid* beállító fájljában, a *squid.conf*-ban lehet megadni, ez a *squid.conf* sablonfájl és a *Squid* beállításait tartalmazó adatszerkezet módosításával vált lehetővé.
Néhány további apróbb, a *Squid* forráskódján végrehajtott módosítás:

- Átfogó függvények meghatározása az *src/protos.h* fájlban
- A `ClientInfo_sb` adatszerkezet típusmeghatározása az *src/typedefs.h* fájlban
- A `ClientInfo_sb` adatszerkezet azonosítójának megadása az *src/enums.h* fájl adatszerkezet-listájában
- A `ClientInfo_sb` adatszerkezet kezdeti értékadása a `memInit()` memóriafoglaló eljárásban, az *src/mem.c* fájlban

Minden módosítást hasonlóan végeztünk el a kódban is, megőrizve az eredeti ügyfél/IP-cím adatbázist. Reméljük, semmit nem rontottunk el.

Áttekintve a módosításainkat, bizonyára feltűnt, hogy minden kódunkat feltételes fordítást előíró `#ifdef SB_INCLUDE ... #endif` szakaszokba helyeztük. Az `SB_INCLUDE` változó akkor kerül megadásra, ha az `--enable-sbclientdb` átadott érték szerepel a *Squid* beállító parancsfájljának parancssorában. Ehhez az *autoconf* segítségével újra kellett fordítanunk a *configure.in* parancsfájlt – előtte elvégeztünk rajta néhány apróbb változtatást.

Összefoglalás

Írásunkban áttekintettük az internetelérés szabályozásának problémáját. Több megoldást is javasoltunk rá, majd részletesebben is ismertettük a *Squid* proxy-kiszolgálóra épülőt, amelyet egyetemünk helyi hálózatában is megvalósítottunk. Tisztában vagyunk vele, hogy megoldásunk nem hozza el a megváltást, valamint nyilván számos hátránya is van, ám egyszerű, rugalmas és teljesen nyílt.
Webes programozónk, *Elmir Mirdiev* mostanában fejez be egy kisebb, *PHP* alapú webhelyet, amelyen keresztül mód nyílik majd a rendszer felügyeletére és a felhasználói statisztikák lekérdezésére. Utóbbiakat a *Squid* naplóiból a *Sarg* rendszerrel állítjuk elő.
További tudnivalók a rendszer forráskódjában találhatók. Webhelyünkről a *Squid 2.5STABLE7* kiadásának teljes módosított változata és a megfelelő foltok külön is letölthetők. Bárki kérdéseit örömmel fogadjuk elektronikus levélben.

Linux Journal 2005. június, 134. szám

Tagir K. Bakirov (batk@mail.ru) rendszergazda a BSPU-nál és első éves doktoranduszhallgató az Ufai Állami Repüléstechnikai Egyetemen. Főként az adatbiztonság, a többügynökös rendszerek és egyéb informatikai témák érdeklik. Szabadidejében sportol, olvas, zenét hallgat és nyelveket tanul.

Vladimir G. Kozlov (admin@bspu.ru) a pedagógiai tudományok doktora, docens, a BSPU vezető rendszergazdája, ahol több informatikai tárgyat is oktat. Elsősorban a *NIX alapú hálózatok, az informatika és az elektronika érdekli. Hobbija a rádiózás (UA9WBZ) a sport és a családja.

KAPCSOLÓDÓ CÍMEK

- www.bspu.ru
- www.netams.com
- www.squid-cache.org
- www.squid-cache.org/Doc/FAQ/FAQ-2.html
- www.squid-cache.org/Doc/Prog-Guide/prog-guide.html
- sarg.sourceforge.net
- www.bspu.ru/squid/squid-2.5STABLE7.sb.tar.gz
- www.bspu.ru/squid/squid-2.5STABLE7.sb.patch

Gentoo másképp

A Gentoo-t legtöbben a fejlesztői munkaállomások egyik vezető terjesztésnek tekintik, de az egyszerű csomagkezelési rendszer miatt olykor jó választás lehet bármely folyamatosan naprakész termelési rendszerben.

Be kell vallanom valamit. ***Gentoo Linuxot*** használok. Kollégáim a különféle Linux felhasználói csoportok találkozóin úgy gondolják lökött vagyok. Mindenki tudja, hogy a ***Gentoo*** forráskód alapú ***Linux***-terjesztés. A ***Gentoo*** általános elképzelés szerint (amit igen nagy mértékben befolyásoltak a terjesztést fejlesztő emberek) olyanoknak való, akik extrém mértékű optimalizálást szeretnének, és igazából csak azoknak felel meg, akik asztali gépen használják. Valójában a ***Gentoo*** meglepő módon rengeteg más feladatra is ideális. Nem kis meglepetésemre néhányan már manapság is ***Gentoo***-t használnak termelési környezetben pontosan ezek miatt az okok miatt.

Sebesség

Minthogy az eredeti ***i386*** processzor leszármazottai binárisan együttműködőek, más ***Linux*** terjesztések (nem beszélve azokról akik ***Microsoft Windowshoz*** írnak programokat) programjuk előrecsomagolt bináris változatát a gyakori ***i386*** rendszerre készítik el kihasználva, hogy így az mindenhol fut. A dolog másik oldala, hogy ezek a terjesztések nem tudják kihasználni azokat az új képességeket amelyeket okos processzorunk nyújthatna, ami azért kár.
A ***Gentoo*** ezzel szemben forrásból-épített terjesztés ahol beállíthatjuk a fordítókapcsolókat amikor rendszerünkhöz alkalmazásokat telepítünk. A ***GCC*** ugyanis lehetővé teszi, hogy meghatározzuk a ***CPU*** típusát, amelyen a kódot futtatni szándékozzuk. A processzortípus meghatározásával (például ***Intel Pentium III*** vagy ***AMD Athlon Thunderbird***) a fordító képes processzorfüggő kódot készíteni, amely (legalábbis elméletben) jobb, gyorsabb gépi kódot eredményez. Gyorsabb tehát a ***Gentoo*** rendszer? Anekdotákon alapuló bizonyítékok elég változóak. Úgy tűnik a ***Gentoo*** rendszerek valamivel gyorsabban futnak mint egy azonosan beállított népszerűbb terjesztés rendszere. Azonban bármilyen apróbb teljesítmény előny azonnal értelmét veszti, ha a rendszer nincs helyesen telepítve beállítva és finomhangolva. Minthogy a legtöbben nem tudjuk hogyan kell ezt csinálni és mivel a ***Gentoo*** olyan nagy mozgásteret nyújt a saját megoldásainknak, ha valami butaságot csinálunk, könnyen elveszíthetjük a némiképp gyorsabb programok előnyét.
Így sebesség tekintetében voltaképp nem igazán számít, hogy forrásból-felépített vagy bináris alapú terjesztést használunk-e. De ha a nagyobb sebesség nem érv a ***Gentoo*** mellett, akkor meg mi értelme van egyáltalán ennek a forrásból telepítés dolognak?

A termelési környezetek általános problémái

Az embereket több módon is felbosszanthatja számítógépük. Amire mi most koncentrálni próbálunk egy újabb keletű probléma, amelybe az új operációs rendszerek két esetben futnak bele. Az első eset, amikor a felhasználó valami olyan eszközkészletet szeretne, ami a terjesztésben nem talál meg. Eredményképpen a felhasználónak magának kell összeállítania azt. A második eset amikor a terjesztésben rendelkezésre álló csomagnak új verziója jelenik meg. Mindkét eset a kulcsa: az egyszerű megoldhatóság. Kérdés, hogy mennyire segít bennünket az operációs rendszer a karbantartást kísérő buktatók során? Legnagyobb meglepetésemre, a ***Gentoo Linux*** úgy tűnik igazán jónak bizonyult ebben a tekintetben. A ***Gentooval*** új csomagokat úgy telepítünk, hogy letöltjük a forrást és befordítjuk. Egy programot szeretnénk? Semmi gond, kiadunk egy utasítást és kicsivel később már fenn is van. Olyan mintha ***Debiant*** használnánk.
Gentoo akkor csillogtatja meg igazán tudását, amikor a felhasználónak újabb verziójú programra van szüksége. Tegyük fel a ***Bluefish HTML*** szerkesztőt használom, ám egy hiba idegesít benne. Könnyen lehet, hogy a ***Bluefish*** újabb verziója már megtalálható a ***Gentoo Portage*** nevű csomagkezelő rendszerében, így lehet, hogy máris lekérhetem a frissítést. A ***Gentoo*** hasznos `etcat` parancsa megmutatja nekünk mit érhetünk el:

```
# etcat versions bluefish
```

Az `etcat` azt mondja, hogy a 0.9-es ***bluefish*** van telepítve viszont már elérhető a 0.12-es is. A ***bluefish*** weblapjáról megtudom, hogy a 0.12-esben már javították a hibát, tehát biztosan frissíteni szeretnék.
A `emerge` parancs megmutatja mi fog történni ha frissítek:

```
# emerge --pretend bluefish
```

Úgy tűnik ennek a ***bluefish*** verziónak szüksége van a ***libpcre*** nevű könyvtárra. A ***Portage*** azt mutatja nekünk,

```
*  app-editors/bluefish :
        [  I] app-editors/bluefish-0.9 (0)
        [   ] app-editors/bluefish-0.11 (0)
        [   ] app-editors/bluefish-0.12 (0)
        [M~ ] app-editors/bluefish-0.13 (0)
```

1. ábra Az etcat eszközzel lekérdezhetjük milyen bluefish verzió áll rendelkezésünkre

```
These are the packages that I would merge, in order:

Calculating dependencies ...done!
[ebuild  N    ] dev-libs/libpcre-4.2-r1
[ebuild     U ] app-editors/bluefish-0.12 [0.9]
```

2. ábra Az emerge – pretend segítségével ellenőrizhetjük a telepítendő program függőségeit

hogy a ***bluefish*** frissítésén felül a ***libpcre*** könyvtárat is le fogja tölteni. Lássunk neki:

```
# emerge bluefish
```

A ***Portage*** először letölti, lefordítja és telepíti a ***libpcre***-t, végül ugyanezt végrehajtja a ***bluefish*** esetében is. Négy perccel később kész is a frissítésem. Hát ez nem volt túl nehéz. Biztosan észrevették, hogy nem a 0.13-as verziót ajánlott fel telepítésre. Ez azért van, mert jelenleg a 0.13-as verzió maszkolt, ezért volt vörössel jelölve. Ebben a felállásban a 0.13 éppen csak kijött, de már van hozzá `ebuild`. Ez az `ebuild` azonban még tesztelés alatt áll, hogy kiderüljön, a program ténylegesen telepítődik-e és nincs-e véletlenül valami durva hiba benne. Amennyiben tényleg szükségünk lenne rá, felülbírálva a ***Portage*** döntését utasíthattuk volna, hogy a 0.13-ast hozza le. Ugyanígy, ha okom van rá, választhattam volna a 0.11-est is. Ez a rugalmasság a ***Gentoo*** legnagyobb ereje.

Csináld magad csomagok

Egy kicsit trükkösebb eset amikor olyan programot szeretnénk felrakni amit nem támogat a rendszer. Az egyik legfontosabb ok amiért terjesztésekben bevezették a csomagkezelést, hogy egyetlen egységes rendszerben láthassák, milyen programok vannak a rendszerre telepítve. Minden egyes programhoz, legyen az alapprogram, rendszerkönyvtár, kiszolgáló program vagy felhasználói alkalmazás, egy csomagot készítenek. Ahogy a csomagot a rendszerre telepítjük, az operációs rendszer rögzíti, hogy mely állományok hova kerülnek, majd a csomag a rendszerre kerül. Ezzel a módszerrel a csomagtól függő más csomagok már tudni fogják, hogy az előkövetelményeik a helyén vannak.
De mi történik, ha újabb verziót akarunk feltenni egy programból, amihez nem készült megfelelő csomag az operációs rendszerünkhöz? Általában ugyanazt az utat kell bejárnunk mint a csomag eredeti készítőjének, kivéve ha a következő két dolog valamelyikét választjuk:

1. Egyedi helyre telepítjük, valószínűleg a ***/usr/local/bin*** könyvtárba, majd biztosítjuk, hogy a régi helyett az új program fusson.

2. Vakon telepítjük a programot a gyökér fájlrendszerre, reménykedve hogy nem nyiffantunk ki valamit közben, és imádkozunk, hogy a később telepítendő dolgok nem fognak semmit felülírni a most feltett dolgok közül.

Gondolkozzunk csak el ezen egy kicsit. Nem találjuk furának ha valaki nem aggódik ilyesmi miatt? Végül is nem éppen ez az, amit a csomagkezelésnek meg kellene előznie? Nem az a kérdés, hogy „lehet-e csomagokat készíteni”, mert erre a válasz „igen, teljeskörűen”, és nem is az, hogy „létre tudjuk-e hozni saját csomagjainkat”. Sokkal inkább azt szeretném tudni, hogy milyen nehéz mindezt végrehajtani. Tegyük fel, hogy fenn van az OS-ból feltett `bogofilter` programunk és a 0.16.1-es verzióhoz való ***rpm*** csomagunk. Hirtelen a `bogofilter` alkotója rádöbben, hogy egy buta, ám igen súlyos hiba rejtőzik a változatban, ezért gyorsan kiad egy 0.16.2-os frissítést.
A probléma ott kezdődik, hogy kénytelenek vagyunk megvárni amíg a terjesztésünk kiadja a saját ***rpm***, ***.deb***, ***.pkg*** vagy más egyéb változatát, ami akár sokáig is eltarthat, így rákényszerülünk, hogy saját változatot készítsünk. Nos, ekkor jönnek a problémák. Elméletileg az ***.rpm*** vagy ***.deb*** csomagok létrehozása egyszerű. „Alapnak egyszerűen csak vegyük a már meglévő 0.16.1 csomagot”. Csakhogy a legtöbb embernek, legalábbis akik nem állnak mágus szinten (de néha még nekik is) elég komoly kihívást jelent az ilyesmi. Saját magunknak kell:

- Letölteni a csomag leírást vagy valahogy kiszednünk egy meglévő csomagból.

- Kézzel letöltenünk és kicsomagolnunk a legújabb ***.tar.gz*** (vagy éppen ami) forrást.

- Át kell alakítanunk a leírásokat (a ***Debian*** esetében a legfrissebb forrásokba) sőt előfordulhat, hogy meg is kell foltoznunk azokat.

- Lehet, hogy meg kell változtatnunk az behúzó-parancsfájlt, hogy helyesen működjön az új verzió esetében is.

- Ezután megpróbálhatjuk létrehozni a csomagot. Ehhez először is le kell fordítanunk, ami valószínűleg jó néhány ***-dev*** csomag feltelepítését jelenti amelyekről azelőtt azt sem tudtuk, hogy léteznek.

- Végül telepítünk és tesztelünk.

Mindez természetesen kivitelezhető ugyan, de a szükséges ismeretek megszerzése igencsak meredek tanulási görbét jelent (különösen az újaknak). Akárhogy is nézzük, ez elég sok munka amit szívesen kihagynánk.

A Gentoo csomagleírásai

Bár alapvetően nem különbözik a fent leírt folyamattól, csomagokat telepíteni egy ***Gentoo*** rendszerre könnyebb. A trükk a ***Gentoo*** egyszerű formátumú `ebuild` csomagleírásaiban rejlik. Ezek lényegében héjprogramok (az ***ebuilds***-al később még foglalkozunk a cikkben). A folyamat közben megadjuk honnan kell beszerezni a forrás tarlabdát. Amikor fordítunk, a Portage letölti a forrást majd

elkezdi kitömöríteni és lefordítani. Mivel héjprogramokról beszélünk, nagy hatékonysággal tudják használni a héjváltozókat. A verziószámot az ***ebuild*** fájlnév értelmezésével nyeri ki majd egy olyan változóba teszi, amit a parancsfájl fel tud használni.
A fenti `bogofilter` példánkban, a csomagállomány (melynek neve ***bogofilter-0.16.1.ebuild***) a következő sort tartalmazza:

```
SRC_URI="http://sourceforge.net/downloads/
➥bogofilter-${PV}.tar.gz"
```

Amikor belefogunk a `bogofiltert` felkészítésebe és fordításába, a ***Portage*** a fájlnév alapján a `$PV` változót 0.16.1-re állítja és a megfelelő ***.tar.gz*** állományt tölti le. Ezek után kicsomagolja, majd a

```
./configure
make
make install
```

parancsok futtatása után az utasításoknak megfelelően felépíti a csomagot. Amennyiben létre szeretnénk hozni a kívánt 0.16.2-es új verziónkhoz tartozó ***ebuild*** állományt a következőket kell tennünk:

```
# cd /usr/portage/net-mail/
# cp bogofilter-0.16.1.ebuild
➥bogofilter-0.16.2.ebuild
# ebuild bogofilter-0.16.2.ebuild digest
```

Feltételezve, hogy a csomagleírásban és a kicsomagolási útmutatókban semmi frissítendő nincsen, mindössze ennyit kell tennünk.
Van még pár dolog amire érdemes odafigyelni. Például, a fentieket valószínűleg inkább a ***/usr/portage*** fa külön másolatán szeretnénk végrehajtani, hogy ne veszítsük el változtatásainkat ha az elsődleges fa esetleg megváltozik. A ***Portage*** közvetlenül is támogatja ezt. Ha kíváncsiak vagyunk, hogyan kell megmutatni a ***Portage***-nek hol tároljuk a saját ***ebuild***-jeinket, nézzük meg a `PORTAGE_OVERLAY` változó leírását a hálózati dokumentációban vagy közvetlenül a ***/etc/make.conf*** állományban. Most már kiadhatjuk a ***Portage***-nek a

```
# emerge bogofilter
```

parancsot és máris megvan az új verziónk.
A ***Gentoo*** valamennyi csomagjának forrás tarlabdájából másolatokat tart fenn a világszerte megtalálható tükrein. Normál esetben a ***Portage*** az egyik ilyen gépről tölti le a forrást. Akkor sincs semmi probléma, ha esetleg valami olyasmit fordítanánk ami nincs fenn a ***Gentoo*** tükrein. A ***Portage*** egyszerűen az eredeti, fejlesztői letöltőoldalt fogja használni.
A ***Portage md5sum*** összegekkel ellenőrzi a letöltött állományok hibátlanságát. A fenti harmadik parancs (`ebuild ... digest`) éppen ezért került oda. Miután letöltöttük a forrást így az md5sum ellenőrzést is rögtön elvégezzük. Mivel most mi hajtjuk végre a verzióugrást, nekünk kell ellenőrizni, hogy biztosan hibátlan-e a letöltésünk. Ezért aztán érdemes először az `ebuild ... unpack` paranccsal megszerezni a letöltést, meggyőződni a hibátlanságáról, és csak ez után feldolgozni a parancsot.
Végül, ha olyan programcsomagot szeretnénk, amit az operációs rendszerünk nem támogat, meg kell írnunk a sajátunkat. A ***Gentooval*** nagyon könnyen készíthetünk saját ***ebuild***-et.

Minden, amit az ebuildről tudni kell

A ***Gentoo*** csomagleírásai ***Bash***-ben készültek. A különféle utasítások függvényekbe kerülnek amelyeket a ***Portage*** hív meg a folyamat során. A főbb elemek:

```
pkg_setup()
src_unpack()
src_compile()
src_install()
pkg_preinst()
pkg_postinst()
```

Ezek sorrendben hívódnak meg. Ha meg akarjuk mutatni a ***Portagenek***, hogyan készítse el a programunkat, írjuk meg a lépésekhez tartozó függvényeket, mindegyiket ellátva némi információval, mint például a korábban említett `SRC_URI`.
A forrásaink fordításához például a következőket használhatjuk:

```
src_compile () {
        ./configure --prefix=/usr
        make
}
```

Ezekben a héjprogramokban az a csodálatos, hogy felüldefiniálható, alapértelmezett változattal rendelkeznek.
Az alapértelmezett `src_compile()` valójában igen hasonló az itt bemutatotthoz, ami sok csomaghoz tökéletes. Tulajdonképpen akár olyan ***ebuild***-et is készíthetünk, amely teljes egészében az alapértelmezett változatokat használja és egyáltalán nem rendelkezik saját függvényekkel.
Néha előfordul, hogy egy csomagot rendszertől függően másképpen szeretnénk beállítani a `./configure`-al. A ***Portage*** ***/etc/make.conf***-ban található és parancssorból felülbírálható `USE` környezeti változója olyan tokeneket tartalmaz, amelyek a rendszerünk testreszabásában és leírásában segíthetnek. Tegyük fel, hogy van egy csomagunk, amit többféleképpen is le lehet fordítani attól függően, hogy szeretnénk-e mondjuk ***X Window System*** vagy ***IPv6*** támogatást. Az `src_compile()` függvényünk a következőképpen nézne ki:

```
src_compile () {
        use X    && conf="${conf} --with-x"
        use ipv6 || conf="${conf} --without-ipv6"
        ./configure --prefix=/usr ${conf}
        make
}
```

Különféle héjprogramozási megoldásokkal találkozhatunk itt. A fenti példánkban ha a rendszerünkön van ***X***, a csomagnak megüzenjük, hogy készítsen ***X*** támogatást.

1. lista ssh2-3.2.9.1.ebuild

```
DESCRIPTION="ssh.com's implementation of SSH2"
SRC_URI="
ftp://mirror.aarnet.edu.au/pub/ssh/
➥ssh-${PV}.tar.gz
ftp://ftp.ssh.com/pub/ssh/ssh-${PV}.tar.gz"
HOMEPAGE="http://www.ssh.com/products"
SLOT="0"
LICENCE="free-noncomm"
KEYWORDS="x86 ~ppc"
RDEPEND="virtual/glibc
        !net-misc/openssh
        >=sys-libs/zlib-1.1.4"
DEPEND="${RDEPEND}
        dev-lang/perl
        >=sys-apps/sed-4"
PROVIDE="virtual/ssh"
IUSE="X ipv6"
RESTRICT="nomirror"
# A csomagot ssh2-nek hívjuk; a forrás
# tarlabdák alakja ssh-x.y.z Ezért felül kell
# írnunk az S-t, hogy kicsomagoláskor a helyes
# könyvtárra mutasson.
S="${WORKDIR}/ssh-${PV}"
# Itt valószínűleg hagyatkozhattunk volna az
➥alapértelmezésre is
src_unpack() {
    unpack ${A}
}
# A nagyszámú configure kapcsoló közül
# az X windows és az Ipv6 befordításának
# lehetőségét kezeljük le itt.

src_compile() {
    local conf
    use X    && conf="${conf} --with-x"
    use ipv6 || conf="${conf} --without-ipv6"
    ./configure ${conf} --host="${CHOST}" \
        --prefix="/usr" \
        --with-ssh-agent1-compat \
        --with-etcdir="/etc/ssh2" \
             || die "configuration failed"
    make || die "compile failed"
}
# ez is szinte azonos az alapértelmezettel,
# de az rc parancsfájl nevét meg szeretnénk
➥változtatni
src_install() {
    make DESTDIR=${D} install
    exeinto /etc/init.d
    newexe ${FILESDIR}/sshd2.rc sshd2
}
```

Amennyiben kiszolgálóról van szó és erre semmi szükségünk, a programunk a felesleges részek nélkül készül el. Az `USE` változókat a parancssorból felülbírálhatjuk, így ha szükséges még pontosabb vezérléssel is dolgozhatunk.

Az `src_unpack()` ugyanígy működik. Amennyiben nem adunk meg semmit, a ***Portage*** nekilendül, az alapértelmezett helyre kitarolja a forrás tarlabdát, belép a könyvtárba és megfelelően beállítja a `$WORKDIR` munkakönyvtár környezeti változót. Másrészről, ha valami szokatlan dolgot is kell csinálni, például egy foltot kell feltenni- magunk is létrehozhatjuk az egyszerű kicsomagoló függvényt:

```
src_unpack () {
        unpack ${A}
        epatch ${FILESDIR}/fixit.patch
}
```

Végül nézzünk egy teljes példát. Volt egy ügyfelem, amely igen sokat használta az ***SSH2*** protokoll ***ssh.com*** által készített változatát. Ezért aztán sok gépre kellett feltelepítenem. Lásd az *1. listát*.
Az ***ebuild*** néhány környezeti változó beállításával kezd:

- `SLOT`: ezeket általában a programkönyvtárakhoz használjuk. Amikor az ***ebuild*** szerzője tudja, hogy ugyanazon csomag különféle verziói lehetnek fenn egy időben a gépen, egy „slot"-szám hozzárendelésével tehet különbséget közöttük. Az egyik rendszeremen fenn van a ***Berkeley DB 1.85***-ös verzió (`SLOT 1`), a 3.2.9-es verzió (`SLOT 3`), a 4.0.14-es verzió (`SLOT 4`) sőt még a 4.1.25_p1-es verzió is (`SLOT 4.1`). Elég sok program létezik, amit a régebbi ***API***-hoz fejlesztettek, és nincs semmi okunk rá, hogy ezeket ne engedjük telepíteni. Amikor aztán a 4.0-ás sorozatban megjelenik egy újabb stabil kiadás, mondjuk a 4.0.17-es verzió, amíg a `SLOT` 4-et használjuk hozzá, a többi verzió eltávolítása nélkül is nyugodtan frissíthetek a 4.0.14-esről. Valójában a ***Berkeley DB*** ennél valamivel összetettebb példa, de jól bemutatja a ***Gentoo*** „slot" elképzelésének erejét. A legtöbb ***ebuildnek*** persze semmi ilyesmire nincs szüksége és a `SLOT="0"` értéket használja.

- `KEYWORDS`: itt jelezhetjük a támogatott architektúrákat. A példában azt mutatom be, hogy ez az ***ebuild*** az ***x86*** sorozaton működőképes és elismerten stabil. A `~ in ~ppc` azt jelzi, hogy ezek maszkoltak. Tudom, hogy a korábbi verziók lefordultak a ***PowerPC*** rendszeren, de nem áll rendelkezésemre egy sem amin kipróbálhatnám, ezért másoknak óvatosnak kell lenniük, ha ezt a verziót választják. A hivatalos ***Portage*** fában az ilyen ***ebuild***-ek néhány hétig ebben az állapotban lehetnek, míg a ***PowerPC*** használók ki nem próbálják azt. Néhány pozitív jelentés után az ***ebuild*** maszkolatlanná válik.

- `DEPEND, RDEPEND`: itt soroljuk fel a függőségeket. Viszonylag teljes nyelvtant és a csomagverziók listáját adhatjuk meg itt. A leggyakoribb módosítók: a `>=`, amely azt jelenti, hogy legalább ilyen verziót kell telepítenünk, általában valamilyen ***API*** miatt amitől a programunk függ; valamint a `!`, amit akkor használunk, ha az adott csomag jelenléte ütközik egy másikkal. Mindkettőt tehát nem telepíthetjük egy időben.

- `RDEPEND`: futásidejű függőségek, ezeket a dolgokat telepítenünk kell a csomag használatához. A `DEPEND` a felépítéshez szükséges függőségeket tartalmazza; a különbség csak akkor mutatkozik meg, ha máshol fordított bináris csomagokat telepítünk.

- `RESTRICT`: itt a *Portage* képességeinek finomhangolását végezhetjük el. Esetünkben, mivel saját készítésű ***ebuild*** csomagról van szó, a `nomirror` segítségével meg tudtam mondani az `emerge`-nek, hogy ne keresgéljen a *Gentoo* tükrein. Ez önmagában még nem jelenti azt, hogy nem használhatom a fejlesztők tükreit. Tulajdonképpen, ha vetünk egy pillantást a `SRC_URI` változóra, látni fogjuk, hogy egy hozzám közel eső tükröt írtam ide, ahonnan a szükséges ***.tar.gz*** állományoka le lehet tölteni.

Ezek után elkezdhetjük felülírni a csomag építését vezérlő függvényeket. A minket érdeklő rész az `src_compile()` függvény lesz. Fogtam a fenti példát és csinosítottam egy kicsit. Láthatjuk, hogy néhány lehetőséget a `USE` változók szabályoznak, míg másokat mi adunk meg, mint például, hogy hol szeretnénk a beállításfájlokat elhelyezni. A ***leállási hiba*** (***die***) üzenetekre nem igazán van szükségünk, viszont jól mutatja, hogy a héjprogramok minden képességét és erejét ki tudjuk használni.
Végül, az `src_install()` függvény helyett is használhattuk volna az alapértelmezettet, de az én rendszeremen az ***/etc/init.d*** állományainak végén nem volt .rc utótag. Ennél is fontosabb, hogy ennek a parancsfájlnak az *OpenSSH*-t kellett lecserélnie a célrendszereken. Ezért biztos akartam lenni benne, hogy az *RC* parancsfájl semmiképp sem ugyanaz, mint ami az *OpenSSH*-t indítja.
A *Portage* gazdag segítő-függvény készlettel rendelkezik amelyek egyszerűsítik az gyakran szükséges feladatokat. Az egyiket ki is használjuk, amikor megmutatjuk hová kell kerülnie az *RC* parancsfájlnak majd ellenőrizzük, hogy végrehajthatónak van-e kijelölve. Ezek után az *ebuild*-et a helyi *Portage* fa másolatunkba helyezve és kiadva az *emerge* parancsot végrehajthatjuk a lépéseket.
Példánk tulajdonképpen csak a felszínt karcolgatja. További részleteket találunk a ➲ *www.gentoo.org/proj/en/devrel/handbook/handbook.xml?part=2&chap=1#doc_chap2* lapon illetve bármely *Gentoo* rendszeren az `emerge --help` kimenetében vagy az `ebuild(1)` és `ebuild(5)` kézikönyv-oldalain.

Több gép karbantartása

Képzeljük el ugyanezeket a problémákat egyetlen gép helyett több tucat kiszolgálóval és munkaállomások ezreivel felszerelt termelési környezet esetében. Nem túl sok operációs rendszert találunk amely segíthet ilyenkor. Sok polcnyi irodalom foglalkozik az infrastruktúra kezelés témájával. Sajnos még mindig elég sok ad-hoc megoldással találkozhatunk. Bár sok gyártó forgalmaz olyan eszközöket, amelyekkel kezdetben tömegével telepíthetünk rendszereket, azonban a későbbi karbantartás feladatát általában a felhasználóra hagyják. Az új verziók problémája nem csak egyetlen gépen jelentkezhet, egész hálózatokat is sújthat.

Tehát hogyan is kerül a *Gentoo* a termelési környezetbe? Itt jön a forrás alapú terjesztés második meglepetése: a *Portage* képes bináris csomagokat is készíteni. Ezzel lehetővé válik, hogy a sarokban felállított egyetlen gép végezze az összes fordítási munkát. Az összes több gép ezeket a bináris csomagokat használhatja, így nem kell maguknak lefordítani a csomagokat. Sokan biztos azt kérdezik magukban „de hát nem éppen ezt teszi a többi *Linux* terjesztés is?" A különbség abban rejlik, hogy itt a helyes csomag-összeállítás kiválasztása a helyi döntésektől függ, és egy újabb verzió alkalmazása egyértelműen olyasmi, amit helyben kell megoldani. A *Gentoo* a z adott rendszercsapatnak olyan eszközöket ad a kezébe, amelyekkel maguk megoldhatják is az ehhez kapcsolódó különféle problémákat.
Egy fordítókiszolgáló alkalmazásával az erőinket és a verziókezelést hatékonyan koncentrálhatjuk, ugyanakkor mégis helyet biztosítunk a helyi sajátosságoknak. Könnyedén összeállíthatunk egy ilyen lépcsős rendszert. Aztán, amikor meg vagyunk elégedve a tesztelt verziókkal, egyszerűen pillanatképet készítünk a bináris csomagokról majd kiosztjuk őket a sorkatona gépeinknek. A *Portage* újabb verziói beépített lehetőséggel rendelkeznek a helyi fájlkiszolgálón készült bináris csomagok letöltésére, így az egész megoldást kulcsrakészen kapjuk.

Összefoglalás

Saját csomagunk készítése vagy egy már létező csomag kézi verzió javítása folyton gondokat okoz. Az tűzhöz közelebb álló csomagkezelők, jóllehet kiforrottabbak, sokkal több nehézség árán képesek csak végrehajtani ezeket a feladatokat. Elméletileg a feladatok egyszerűek. Legnagyobb meglepetésemre, hiszen ezt az oldalát nem is reklámozzák a dolognak, a *Gentoo* eszközeivel magunk is könnyedén megoldhatjuk ezeket a problémákat.

Köszönetnyilvánítás

A szerző szeretne köszönetet mondani ***Stephen Whitenak*** a ***Adelaide Egyetemről*** és ***Andrea Barisaninak*** a ***Trieszti Egyetemről*** hogy segítettek kidolgozni az itt bemutatott termelési környezetben is használható *Gentoo Linuxra* vonatkozó ötleteket. Ezen kívül voltak olyan kedvesek és átnézték a cikket, ahogy azt ***Pia Smith*** a ***Linux Australia***-tól, ***Jeff Waugh*** a *GNOME*-tól, és ***Craige McWhirter*** a ***Sydney Linux Users Group***-tól valamint ***Wade Mealing*** a *Gentoo Server Projectből* is tette.

Linux Journal 2005. február, 130. szám

Andrew Cowie készíti az Operational Dynamics (www.operationaldynamics.com), operatív és infrastruktúra mérnöki tanácsadással foglalkozó oldalt. Szervezeteknek segít értékesíteni a technológiájukat az emberekre és az emberek körüli folyamatokra koncentrálva. Valószínűleg éppen amiatt keres állandóan egyszerűbb megoldási lehetőségeket. Elérhető az andrew@operationaldynamics.com címen vagy az irc.freenode.net AfC néven.

Adatbázisok és naptárak

Az iCalendar szabványt már megismertük, most itt az ideje, hogy az adatokat adatbázisból vegyük, és a naptárakat futásidőben állítsuk elő.

A múlt hónapban folytattuk az ismerkedést az *iCalendar* szabvánnyal, amelyre alapozva a programok naptár- és találkozóadatokat cserélhetnek egymással. Mint láttuk, egy *iCalendar* fájl egy vagy több eseményt és feladatot, tennivalót tárolhat. Ha a fájlt elérhetővé tesszük egy http kiszolgálón, például *Apache*-on keresztül, akkor bárki számára terjeszthetjük, aki az *iCalendar* kezelésére alkalmas programmal rendelkezik – ilyen például a *Mozilla Sunbird*. Amint a múlt hónapban láttuk, ennél eggyel tovább is léphetünk, és az *iCalendar* fájlt dinamikusan, *CGI* programmal is előállíthatjuk.

A múlt hónapban bemutatott programokat ugyan bizonyos körülmények között lehet hasznosítani, az minden webes fejlesztő számára egyértelmű, hogy a dátum- és időadatokat a programon belül tartani finoman szólva ostobaság volna.

Az ilyen adatok tárolását a legjobban relációs adatbázisban oldhatjuk meg, például *PostgreSQL* segítségével. Relációs adatbázis alkalmazásával könnyebb biztosítani a bevitt adatok érvényességét, továbbá gyors és rugalmas hozzáférést kapunk az elmentett adatok egy részéhez vagy egészéhez. További előny, hogy ha a naptáradatokat adatbázisban tároljuk, akkor ugyanazt a forrást használva egy-egy naptárfájlból több változatot is elő tudunk állítani.

Ebben a hónapban egy egyszerű, web alapú programra fogunk példát nézni, mely relációs adatbázisból veszi a naptáradatokat, ezek alapján előállít egy *iCalendar* adatfájlt, ami viszont alkalmas lesz az *iCalendar*-ral együttműködő programokba – mint a már említett *Mozilla Sunbird* – történő importálásra.

A tábla megadása

Ha a naptáradatokat relációs adatbázisban kívánjuk tárolni, meg kell adnunk legalább egy táblát. Ennek oka az, hogy a relációs adatbázisok minden adatot – sokszor még a beállításokat és az állapotadatokat is – kétdimenziós táblázatokban tárolnak, amelyekben az oszlopok határozzák meg az egyes mezőket, míg egy-egy sor egy-egy rekordot tartalmaz. *PostgreSQL* alatt például a következő módon tudunk megadni egy egyszerű táblát:

```
CREATE TABLE Events (
  event_id        SERIAL    NOT NULL,
  event_summary   TEXT      NOT NULL
    CHECK (event_summary <> ''),
  event_location   TEXT      NOT NULL
    CHECK (event_location <> ''),
  event_start     TIMESTAMP NOT NULL,
  event_end       TIMESTAMP NOT NULL,
  event_timestamp TIMESTAMP NOT NULL
    DEFAULT NOW(),
  PRIMARY KEY(event_id)
);
```

A fenti tábla hat oszlopot tartalmaz. Az első az event_id (eseményazonosító), típusa `SERIAL`. Ha az `event_id`-nek nem adunk kifejezett értéket, amikor hozzáadunk egy-egy sort a táblához, akkor a *PostgreSQL* önműködően vesz egy legfeljebb 2^{31} értékű egész számot. A *PostgreSQL* nagyobb felső határ megadását vagy az 1-es értékhez való körbeforgást is lehetővé teszi, erről részletesebb tájékoztatás a leírásában található.

Az `event_id` oszlop tartalma egyedileg azonosítja a tábla sorait, ezt a `PRIMARY KEY` (elsődleges kulcs) kulcsszó (vagyis inkább kulcskifejezés) tudatja az adatbázissal. Ez nemcsak a rekordok lekérdezéséhez szükséges oszlopot adja meg a többi adatbázis-programozónak, de az értékek egyediségét és az oszlop indexelését is biztosítja.

Szintén automatikusan kap értéket az `event_timestamp` (esemény időbélyeg). A megadásból látható, arra is van mód, hogy explicit értéket adjunk neki (ezt meg is fogjuk tenni), de alapértelmezett értéke az aktuális idő. Az általános az, hogy ha ilyen formában adunk meg egy oszlopot, akkor explicit értéket csak kivételes alkalmakkor kap, és a legtöbb esetben a *PostgreSQL*-re bízzuk feltöltését az aktuális dátummal és idővel.

Megjegyezném, hogy az `event_summary` (esemény összegzése) és az `event_location` (esemény helye) egyaránt `TEXT` típusú (vagyis ezek végtelen hosszúságú, szöveges mezők), míg az `event_start` (esemény kezdete), az `event_end` (esemény vége) és az `event_timestamp` (esemény időbélyege) `TIMESTAMP` (időbélyeg) típusú; az *SQL*-ben ez a szabványos megoldás dátum és idő megadására.

1. kódrészlet db-calendar.py

```
#!/usr/bin/python
# A CGI modul beemelése
import cgi
import psycopg
from iCalendar import Calendar, Event
from datetime import datetime
from iCalendar import UTC # időzóna
# Az esetleges hibák naplózása
import cgitb
cgitb.enable(display=0, logdir="/tmp")
# content-type fejrész küldése
print "Content-type: text/calendar\n\n"
# Naptár objektum létrehozása
cal = Calendar()
# Milyen termék hozta létre a naptárat?
cal.add('prodid',
        '-//Python iCalendar 0.9.3//mxm.dk//')
# Az RFC 2445-nek a 2.0-s változat felel meg
cal.add('version', '2.0')
# Adatbázis-kapcsolat létesítése
db_connection =
  psycopg.connect('dbname=atf user=reuven')
db_cursor = db_connection.cursor()
db_cursor.execute
  ('''SELECT event_id, event_summary,
  ➥event_location,
            event_start, event_end,
            ➥event_timestamp
        FROM Events
    ORDER BY event_start''')
result_rows = db_cursor.fetchall()
for row in result_rows:
    # Esemény létrehozása
    event = Event()
    # Az eseményazonosító megadása
    event['uid'] = str(row[0]) + 'id@ATF'
    # Leírás és helyszín megadása
    event.add('summary', row[1])
    event.add('location', row[2])
    # A dátumok átalakítása
    event.add('dtstart', datetime(tzinfo=UTC(),
              *row[3].tuple()[0:5]))
    event.add('dtend', datetime(tzinfo=UTC(),
              *row[4].tuple()[0:5]))
    event.add('dtstamp', datetime(tzinfo=UTC(),
              *row[5].tuple()[0:5]))
    # Kapjon kiemelt fontosságot!
    event.add('priority', 5)
    # Az esemény hozzáadása a naptárhoz
    cal.add_component(event)
# A naptár utasítása önmaga leképezésére iCalendar
# fájlként, majd a fájl átadása a HTTP-válaszban.
print cal.as_string()
```

A tábla összes oszlopánál szerepel a `NOT NULL` (nem nulla) kifejezés, vagyis ezek nem kaphatják az ***SQL*** esetében a meghatározatlant jelentő `NULL` értéket. A `NULL` nem azonos az igazzal vagy a hamissal, ami a kezdők esetében sokszor félreértéseket okoz. A `NULL`-t mint ismeretlen, meg nem adott értéket kell kezelni – így talán könnyebb. Bár a `NULL` értékek segítségével könnyen meg tudjuk különböztetni a hamis és az ismeretlen értékeket, használatukat érdemes a lehető legszűkebb körre korlátozni. Sőt, általában azt szokták tanácsolni, hogy alapesetben minden oszlopot `NOT NULL`-ként, vagyis nem nullként adjunk meg, és csak akkor engedélyezzük a `NULL` értékek használatát, ha az valóban szükséges.

Felhívnám még a figyelmet a két szöveges oszlop megadására (`event_summary` és `event_location`), melyek egyrészt a `NOT NULL` kulcsszóval vannak ellátva, másrészt egy további, az üres karakterláncok bevitelét megakadályozó ellenőrzés tárgyát képezik. Az, hogy ezek az ellenőrzések helyénvalók-e, a tényleges alkalmazási környezettől függ. Előfordulhat, hogy valamiért inkább engedélyezni akarjuk a `NULL` értékek használatát, és az összegzésben vagy a hely megjelölésében szereplő üres karakterláncok sem okoznak gondot.

Bár a fenti táblamegadást csak egyszerű példának szántam, gondoljunk bele, milyen jó volna a helyszíneket külön táblában tárolni, például egy `location_id` (hely azonosítója) és `location_name` (hely neve) oszloppal, majd a szöveges `event_location` oszlopot a `location_id`-vel helyettesíteni. Így egységesíteni tudnánk a helyszínneveket, vagyis maga az adatbázis is egységesebbé válna, illetve arra is módunk nyílna, hogy kikeressük az adott helyszínen várható eseményeket.

Amikor végeztünk a tábla megadásával, hozzáadunk néhány indexet. Az indexek révén az adatok lekérdezése gyorsabb lesz a normál lekérdezéseknél; ennek az `INSERT` (beillesztési) műveletek megnövekedett végrehajtási idejével fizetjük meg az árát. Nézzük a megadásokat:

```
CREATE INDEX event_location_idx
  ON Events(event_location);
CREATE INDEX event_start_idx
  ON Events(event_start);
CREATE INDEX event_end_idx
  ON Events(event_end);
```

Új adatok beillesztése

Elkészült a tábla és az indexek, kezdjük el eseményekkel feltölteni az adatbázist. Az események beillesztésére az `INSERT` parancs szolgál, melynek írásmódja a következő:

```
INSERT INTO Events
  (event_summary, event_location,
   event_start, event_end)
VALUES
  ('Március idusa', 'Mindenhol',
 '2005-March-15 00:00', '2005-March-15 23:59:59')
```

Mint látható, a fenti `INSERT` utasításban az `Events` tábla hat oszlopából csak négy szerepel. Ha megvizsgáljuk az új sor tartalmát, a következőt találjuk:

```
atf=# select * from events;
-[ RECORD 1 ]---+---------------------------
event_id        | 1
event_summary   | Március idusa
event_location  | Mindenhol
event_start     | 2005-03-15 00:00:00
event_end       | 2005-03-15 23:59:59
event_timestamp | 2005-04-04 01:20:15.575032
```

Mint látható, az `event_id` (mely, mint emlékszünk, `SERIAL` típusú) automatikusan kapott egy 1-es értéket, az `event_timestamp` pedig megkapta értékül a beillesztési kérés végrehajtásának időpontját.
Könnyű elképzelni, hogyan hívnánk meg az `INSERT` utasítást valamilyen web alapú programból, például *CGI*-n, esetleg valamilyen fejlettebb rendszeren keresztül, mint a *mod_perl* vagy a *Zope*. Hozzátenném, az adatbázisba befutó adatok beérkezésének módjával sem kell foglalkoznunk, különösen, ha megadtuk a megfelelő megszorításokat. Nyugodtan feltételezhetjük, hogy az adatbázisban található adatok megbízhatóak, és a kiszolgáló eleve elutasította az előírásoknak meg nem felelő beviteleket.

Dinamikus iCalendar fájl létrehozása

Ha már van néhány eseményünk az adatbázistáblában, megpróbálhatjuk lekérdezni őket egy *CGI* programból. A program kimenete *iCalendar* formátumú lesz, vagyis az *iCalendar* ügyfelek is hozzáférhetnek majd az adatokhoz. Az *1. kódrészlet* egy ilyen programot tartalmaz, ez egyébként a múlt hónap *dynamic-calendar.py* programjának módosított változata. Mint a múltkor már utaltam rá, a programot nem kis részben azért *Pythonban* írtam, mert viszonylag szűkében állunk az *iCalendar* formátumú fájlok előállítására használható moduloknak. Szerencsére *Python* alá van ilyen modul, és én nem voltam rest kihasználni ennek előnyeit. Mint az 1. kódrészletből is látható, semmi bonyolultra nem kell számítani. Beemelünk néhány modult, létrehozunk egy naptár objektumot, majd beillesztjük az *iCalendar* szabvány által megkövetelt mezőket, megadva a naptár forrását. Ez után csatlakozunk egy a feltételezésünk szerint a helyi gépen futó *PostgreSQL* kiszolgálóhoz. Bár *Python-PostgreSQL* párosításhoz több adatbázis-csatoló is létezik, én a *psycopg*-t használom, mely gyors és üzembiztos. A *PostgreSQL*-hez a *psycopg* segítségével a következő írásmódot alkalmazva kapcsolódhatunk:

```
db_connection = psycopg.connect
                ('dbname=atf user=reuven')
```

A fentiek szerint az adatbázis neve `atf`, a felhasználónév pedig `reuven`. További lehetőség a kiszolgáló és valamilyen jelszó megadása, amire inkább akkor lehet szükség, ha éles rendszeren dolgozunk.
Miután csatlakoztunk az adatbázishoz, veszünk egy mutatót, ezzel tudjuk elküldeni a kéréseinket és lekérdezni az eredményüket:

```
db_cursor = db_connection.cursor()
```

Kezünkben a kurzorral immár tudunk *SQL*-lekérdezéseket küldeni az adatbázisnak; ezeket a *Python* „három idézőjeles" írásmódját kihasználva tettem könnyebben olvashatóvá. A következő lépés az eredmények lekérése. Ha nagyobb számú, akár több száz találatra számítanánk, akkor célszerűbb volna egyenként, esetleg csoportokban lekérni őket, ez esetben azonban tudjuk, hogy naptárunk mindössze néhány eseményt tartalmaz, ezért a `fetchall()` hívással egyetlen lépésben lekértem az összest.

```
result_rows = db_crsor.fetchall()
```

A `result_rows` (eredmény sorok) minden egyes eleme egyegy sor a *PostgreSQL* adatbázisból. Ha tehát egy `for` ciklussal végiglépkedünk a sorokon, akkor megkapjuk a találat egyes elemeit.
A legtöbb esetben az egész eljárás rutinműveletnek számít, ám amikor dátumokkal és időpontokkal dolgozunk, már több figyelmet kell szentelnünk az adatkezelésnek – márpedig a dátumok a naptár fontos elemei. A gond az, hogy a `psycopg` az *eGenix.com* nyílt forrású mxDateTime modulját használja, amellyel rendkívül könnyű a dátumokat kezelni. Csakhogy az *mxm iCalendar* modulja a Python datetime modulját használja, ami viszont eltérő. Le kell tehát kérdeznünk az egyes dátumokat (az egyes események kezdő és záró dátumát), *mxDateTime* példányból *datetime*-megfelelő tuple-é kell alakítanunk őket, majd a tuple felhasználásával elő kell állítanunk egy *datetime*-példányt, amelyet már át tudunk adni az `event.add`-nek:

```
event.add('dtstart', datetime(tzinfo=UTC(),
event.add('dtend', datetime(tzinfo=UTC(),
          *row[3].tuple()[0:5]))
```

A fenti három sornyi kódban a `datetime()` második átadott értéke pontosan az imént felvázolt eljárást hajtja végre. A kapott sorból vesz egy oszlopot, majd tuple-é alakítja. Ezután fogjuk a sorozat egy darabját (a *Python* kényelmesen használható [0:5] írásmódjával), ezzel hozzájutunk a `tuple()` által visszaadott elemek egy részhalmazához.
A `datetime()`-nak viszont nem lehet sorozatot átadni, csak különálló elemek sorozatát. Másként fogalmazva, a `datetime()` több számot vár, és nem hivatkozást vagy mutatót számok egy listájára. A tuple-t a *Python* * operátorával bontjuk önálló elemekre. Végül, az élesebb szemű olvasók bizonyára észrevették, hogy a `tzinfo` átadott érték még a tuple elemei előtt szerepel, ennek oka az, hogy *Pythonban* a nevesített átadott értékeket még a * operátor előtt kell megadni.

További lehetőségek

A *db-calendar.py* futtatásával az *iCalendar* szabványnak tökéletesen megfelelő, a *Sunbird* vagy más naptárprogram alá gond nélkül importálható fájlt kapunk. Emellett, ha elvégzünk egy egyszerű módosítást az `Events` táblán, biztosíthatjuk, hogy a naptárra feliratkozó felhasználók mindig a legújabb változatot kapják meg.

Ennél tovább is mehetünk, érdemes például a *db-calendar.py*-t úgy módosítani, hogy kimenetében csak bizonyos események szerepeljenek. Lehetséges például, hogy megelégszünk a jövőbeli események átadásával, és nem terheljük mások naptárát (és sávszélességét) a múlt történéseivel. Ilyenkor elég egy egyszerű `WHERE` utasítást adnunk az *SQL*-lekérdezéshez, és a már megtörtént eseményeket könnyedén kiszűrhetjük.
Ennél érdekesebb lehetőség a különféle naptárbeli csoportok és elérési szintek elkülönítése. A *HTTP* támogatja a felhasználónév és jelszó alapú hitelesítést, és bár a *Sunbird* jelenleg nem teszi lehetővé az ilyen jellegű védelem alkalmazását, a jövőben várható ez irányú továbbfejlesztése (ahogy a többi programé is). Figyelembe véve, hogy a *CGI* programok könnyen meg tudják határozni a hitelesített *HTTP* kérésekhez tartozó felhasználónevet, talán nem túlzás azt mondani, hogy a *db-calendar.py* a különböző felhasználók számára különböző kimeneteket is elő tudna állítani, a hozzárendelt engedélyek és szerepek alapján.
Végül utalnék rá, hogy bár az elmúlt hónapokban az *iCalendar* formátumra összpontosítottunk, nincs semmi oka annak, hogy az adatbázis tartalmát kizárólag *iCalendar* fájlként tegyük elérhetővé. Miért ne tehetnénk meg például, hogy az *iCalendar* mellett a jó öreg *HTML* formátumban is megjelenítjük az eseményeket? *HTML* táblázatok segítségével roppant egyszerű volna a feladat megoldása – ez is kiváló példája annak, hogy relációs adatbázisra támaszkodva az adatok különféle formátumokban való megjelenítése nem probléma, sokkal inkább lehetőség.

Összefoglalás

Röviden áttekintettük, hogyan alkalmazhatunk adatbázist az események adatainak tárolására, amelyek alapján végül *iCalendar*-megfelelő fájlt állítottunk elő. Adatbázis használatával nemcsak a tárolt adatok megbízhatóságát növelhetjük, de könnyen és gyorsan állíthatunk elő dinamikus, az *iCalendar* formátum olvasására képes programok által kezelhető fájlokat.

Linux Journal 2005. július, 135. szám

Reuven M. Lerner (➲ http://www.lerner.co.il/atf) Nyílt forrású programokra, valamint web- és adatbázis-alkalmazásokra szakosodott tanácsadó. Könyve, a Core Perl, 2002 januárjában jelent meg a Prentice Hall gondozásában. Reuven feleségével és lányaival nemrég költözött Chicagóba.

KAPCSOLÓDÓ CÍMEK

- ➲ www.postgresql.org
- ➲ www.python.org
- ➲ www.mxm.dk/products/public/ical/downloads
- ➲ techdocs.postgresql.org/techdocs/faqdatesintervals.php
- ➲ initd.org/projects/psycopg1
- ➲ www.mozilla.org/projects/calendar/download.html
- ➲ www.ietf.org/rfc/rfc2445.txt

FreeBSD – a szomszéd vár (10. rész)
A vár őrei

Egy kiszolgáló gép elsődleges feladata, hogy a kliensek kéréseit kiszolgálja, s eközben meg kell különböztetnie a klienseket a támadóktól. A BSD vár önmagában elegendő bizonyos fokú védelemre, azonban ezt érdemes kibővíteni.

Betörési pontok

Több támadási típusra kell felkészülnünk, amelyek gyakran szolgálnak betörési útként különféle támadóknak, legyenek humán betörők, vagy jól megírt programok (férgek vagy vírusok).

A ***szolgáltatásmegtagadás (Denial of Services, DoS)*** célja, hogy elfogyassza a gépünk egyes erőforrásait, amelyek fontosak az üzemszerű működéshez. Többnyire nyers erőt felhasználva annyi kérést zúdítanak a célzott gépre, hogy az képtelen lesz a kiszolgálásra, esetleg össze is omlik. Kisebb arányt képviselnek azok a támadások, amelyek egy szolgáltatás programozási hibáját kihasználva egyetlen hálózati kérést küldenek csupán, s ez már elegendő arra, hogy akár a kiszolgáló is a padlóra kerüljön.

Ez utóbbi esetben az adott szolgáltatás vagy a rendszermag hibáját kijavítva a ***DoS*** lehetősége megszűnik. Az előbbi esetben érdemes korlátozni az egyes szolgáltatások által használható erőforrásokat, így a ***DoS*** csak ideiglenesen jár sikerrel.

A felhasználói jog szerzése sokkal gyakoribb és elterjedtebb támadási mód. Sok olyan szolgáltatás fut még, amely a felhasználói adatokat (név és jelszó) titkosítás nélkül küldi és fogadja (***FTP***, ***POP3***, stb). Ezeket az információkat a hálózati forgalom figyelésével könnyedén le lehet jegyezni, s később felhasználni. Egy bizonyos felhasználói számot túllépve a rendszergazda már nem lesz képes kideríteni, hogy valóban az adott felhasználó lépett be egy késő esti órában dolgozgatni, vagy sikeresen ellopták az adatait.

Előfordulhat, hogy a root felhasználó nevében sikerül belépnie a támadónak, akár egy hibás program biztonsági hiányosságát kihasználva, akár a jelszó figyelésével. Számolnunk kell azzal, hogy a rendszergazdai jogok birtokában könnyedén eltávolíthatja a betörés és az adatlopás nyomait. Ritka eset, amikor a támadók összeomlásra késztetik a kiszolgálót magát, gyakoribb, hogy hátsó ajtót hoznak létre, s azon már észrevétlenül képesek közlekedni. A hátsó kapuk gyakorta más gépek feltöréséhez adnak rejtőzési lehetőséget a profi adatgyűjtők számára.

Alapvető biztonság

Ha nem tesszük biztonságossá a root hozzáférést, akkor felesleges dolgoznunk a többi alrendszer biztonságossá tételén, a támadóknak tálcán kínáljuk rendszerünk leginkább áhított gyenge pontját. A legtöbb kiszolgáló esetén nem fontos a root hozzáférést konzolon kívül hozzáférhetővé tenni, ezen túllépve még akár a konzolon is megtilthatjuk, s így csak egy meghatározott felhasználón át tudunk rendszergazdai jogokhoz jutni, mégpedig a su parancsot használva. A távoli hozzáférést a /etc/ttys állományban tudjuk korlátozni, így az általános programokat tekintve (***telnet***, ***rlogin***, stb.) már kizártuk azon próbálkozásokat, amelyek egyenesen a root felhasználó jogaival szeretnének bejutni. Egyéb esetben a programok és szolgáltatások saját beállításaival tudunk még sikereket elérni: például az ***SSH*** esetén a ***/etc/ssh/sshd_config*** állományban a `PermitRootLogin` változó értékét állítsuk **NO** állapotra.

Ha sikeresen megtiltottuk a közvetlen rendszergazdai belépéseket, akkor sajnos eléggé jól elvágtuk magunkat a szervertől hálózaton át: ugyanis akárki nem lehet a `su` parancs segítségével rendszergazdai jogok birtokosa:

```
tomy66@szerver ~ $ su -
su: Sorry
```

A `su` parancs ugyanis azonnal visszautasítja a root jogok felé irányuló kéréseket, még a jelszót sem kéri el. A ***FreeBSD*** külön csoportban kezeli azon felhasználókat, akik képesek rendszergazdai jogokat szerezni: a `wheel` csoport szolgál erre a célra:

```
tomy66@szerver ~ $ grep wheel /etc/group
wheel:*:0:root,auth.gabor,franko
```

Ha az aktuális felhasználó nincs a megadott csoportban, akkor sajnos (illetve szerencsére) nem lehet

belőle rendszergazda az adott gépen. A csoportba tartozó felhasználókat viszont jelszó ellenében már közel engedi a tűzhöz:

```
auth.gabor@szerver ~ $ su -
Password:
```

Ezzel a módszerrel az adott felhasználó(k) képesek lesznek belépni a kiszolgáló gépre, s a saját jelszavukat megadva, majd a root jelszót megadva már rendszergazdai joggal bírnak. Sajnos azonban mind a két jelszót be kell gépelni a billentyűzeten, így az kifigyelhető és felhasználható betörési célokra. Ennek okán érdemes a ***SSH*** esetén az adott felhasználó jelszavát megszüntetnünk (nem üresre állítanunk!). Ennek módja egyszerű, a `vipw` parancs segítségével szerkesztjük a ***passwd*** állományt (gyakorlatilag a /etc/master.passwd állományt látjuk ekkor), majd a megfelelő felhasználónévnél az alábbi sort

```
auth.gabor:$1$tLlpY8GZ$HOPQAc3eqNy1adsR1XmqB1:1001:
➥1001::0:0:Auth
Gábor:/home/tanar/auth.gabor:/usr/local/bin/bash
```

így módosítjuk:

```
auth.gabor:*:1001:1001::0:0:Auth
➥Gábor:/home/tanar/auth.gabor:/usr/local/bin/bash
```

A beírt és kódolt jelszó soha nem lesz azonos a * karakterrel, így tetszőleges jelszóval sem lehet majd belépni ezzel a felhasználóval. Kell készítenünk egy kulcspárt (egy privát és egy publikus kulcsot), mégpedig az `ssh-keygen` parancs segítségével. A publikus kulcsot hozzá kell másolnunk a kiszolgáló gép felhasználójának ***.ssh*** könyvtárában lévő ***authorized_keys*** állományához. A kliens gép ***.ssh*** könyvtárába pedig a privát kulcsot kell őriznünk, ***identity*** néven.
Ha be szeretnénk lépni ***SSH***-n keresztül a kiszolgáló gépre, akkor az ***SSH*** nem lesz képes jelszavas azonosításra, hiszen bármilyen jelszót is írnánk be, az nem lesz megfelelő. Ellenben a kulcspáron alapuló hitelesítés működik, így ha az ***identity*** állományban lévő privát kulcsból előállított publikus kulcs megtalálható a kiszolgáló ***authorized_keys*** állományában, akkor jelszó és kérdés nélkül be fog engedni. A feladatunk mindössze annyi, hogy ne engedjük ki a kezünkből a privát kulcsot:

```
$ ssh auth.gabor@szerver
Last login: Thu Jun 30 09:31:28 2005 from kliens
szerver:~ $
```

Mindig csak annyi szolgáltatást futtassunk, amennyi szükséges; s mindig figyeljünk oda az újabb verziókra, hibajavításokra. Egy régen frissített szolgáltatás akár egy széles és egyenes út lehet a támadók számára, hiszen lehetnek benne biztonsági hiányosságok. Figyeljük rendszeresen a biztonsági értesítőket, hiszen órákon is múlhat rendszerünk épsége. Ha megállapítottuk, melyek a szükséges szolgáltatások, akkor lehetőleg mindegyiket tegyük egy ***jail*** által létrehozott „homokozóba" (***sandbox***). Ezek a „homokozók" nem jelentenek túl nagy biztonságot, azonban a támadók először ezekbe kerülnek, és innen is tovább kell jutniuk a rendszerünk belső részei felé. Törekedjünk minél több ilyen réteg elkészítésére, így egyre több időt kell a támadóknak a rendszer megismerésével és feltörésével eltöltenie, így a mi rendszerünk egyre kevésbé lesz vonzó számukra.
A ***FreeBSD*** operációs rendszer sok szolgáltatást alapból homokozóban futtat (***ntalkd***, ***fingerd***, stb.), így ezekkel nem is kell törődnünk. További hibaforrás a ***SUID/SGID*** bitekkel ellátott programok telepítése, mivel az ezekben előforduló hibák automatikusan rendszergazdai jogokat biztosítanak a felhasználók számára. Jól jegyezzük fel, hogy ezen programok közül mennyit telepítünk fel, illetve ezek hol találhatók: ha a méretük megváltozik, akkor annak komoly következményei lehetnek.

Kiterjesztett biztonság

Amint az alaprendszert biztonságossá varázsoltuk, még közel sem végeztünk a munkával. A ***FreeBSD*** programtelepítésének alapja a ports adatbázis, amely már 12.000 programcsomag felett tart. Éles rendszer esetén több problémával is szemberülhetünk a ports használatával, amelyek közül az egyik nagyon lényeges: nem értesülünk az újabb csomag kibocsátásának okáról. Egy egyszerű ***portupgrade*** végrehajtásakor az összes változott csomagot frissíthetjük, ám a ***„Ha működik, ne javítsd!"*** aranyszabályt ezzel csúfosan megszegjük. Többnyire nincs probléma, mivel az újabb programcsomag kompatibilis a régebbivel, ha azonban akadtak kisebb változtatások, akkor hamar és sürgősen kellett a hibát kijavítani (vagy visszatenni a régebbi verziót).
Ezen a problémán tud segíteni a ***security/portaudit*** csomag, amely a ***periodic*** programon keresztül rendszeresen elvégzi a telepített csomagok összevetését a rendelkezésre álló csomagokkal, s jelzi a csere szükségességi fokát.
A program telepítése után azonnal kérhetünk egy jelentést a frissítésre váró programokról, egyszerűen a

```
$ /usr/local/sbin/portaudit -Fda
```

parancsot kell kiadnunk. Ezek után a program egy új adatbázis letöltésével kezdi a munkáját, amely a frissítésre váró programok adatait tartalmazza:

```
auditfile.tbz           100% of   26 kB   27 kBps
New database installed.
Database created: 2005 Jún 30 Csü 18:40:13 CEST
```

Ezt követi azon csomagok felsorolása, amelyek esetén biztonsági okból javasolt a frissítés:

```
Affected package: ruby-1.6.8.2004.07.28_1
Type of problem: ruby -- arbitrary command
➥execution on XMLRPC server.
Reference: <http://www.FreeBSD.org/ports/
➥portaudit/594eb447-e398-11d9-a8bd
➥-000cf18bbe54.html>
```

A felsorolt programok kapcsán kettő lehetőségünk adódik: frissítjük vagy eltávolítjuk őket. Egyéb esetben a rendszerünk biztonsági oldalról hibás és hiányos marad, s nyitva

hagyjuk a kaput a támadók számára. Érdemes a program által kiemelt ***URL*** meglátogatása, s így bővebb információkat is tudunk szerezni a biztonsági fenyegetettségről. Esetleg létezik kerülő megoldás is (***workaround***), amely egy kissé kitolhatja a kockázatos frissítés idejét, de ez többnyire a szolgáltatás minőségromlásával jár.
Mivel a program minden nap lefut legalább egyszer, a fenti műveletekről levelet kapunk (ha olvassuk a root felhasználónak írandó leveleket :), és így a *„security run output”* tárgyú levélben olvashatunk a telepített csomagok biztonságosságáról.

BSD Process Accounting

Sok felhasználó esetén célszerű a futtatott programok nyomkövetése, így több napra, hétre visszamenőleg képesek leszünk lekérdezni, hogy melyik felhasználó – melyik programot – mennyi ideig használta.
A *Process Accounting* bekapcsolásához egyszerűen a */etc/rc.conf* állományba bele kell tennünk a

```
accounting_enable="YES"
```

sort, s ezzel a következő indításkor már futni is fog a szolgáltatás. Az azonnali bekapcsolásához az

```
$ accton
```

parancsot tudjuk használni. Ettől az időponttól kezdődően nyomon tudjuk követni felhasználóink tevékenységét:

```
$ lastcomm auth.gabor
[...]
kwebdesktop auth.gabor 2.22 secs Thu Jun 30 18:43
sh          auth.gabor 0.00 secs Thu Jun 30 18:41
[...]
```

Érdemes a *ports* adatbázis *security* kategóriáját figyelmesen átnézni, és a szükséges programokat használni, így ezzel is csökkentjük a biztonsági fenyegetettségeinket.

Auth Gábor (auth.gabor@enaplo.hu)
Egy pécsi középiskolában informatikát és programozást oktat. Tíz éve botlott először a UNIX rendszerekbe, 7 év Linux használat után kapta el a FreeBSD lázat, amiből máig nem tudott kigyógyulni.

KAPCSOLÓDÓ CÍMEK

A FreeBSD projekt honlapja: ➲ http://www.freebsd.org

A magyar FreeBSD honlap: ➲ http://www.freebsd.hu

A magyar BSD honlap: ➲ http://www.bsd.hu

A kézikönyv magyar fordítása
➲ http://www.enaplo.hu/FreeBSD/handbook/

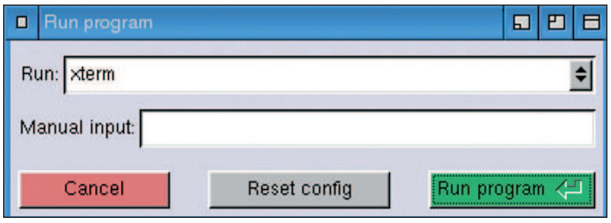
Programfejlesztés az eFluid segítségével

Sorozatunk utolsó fejezetében egy egyszerű programot fogunk elkészíteni az eFltk grafikus fejlesztőeszközének segítségével.

Gyakran előfordul, hogy bizonyos ablakkezelő alkalmazások használata közben egyszerű módon szeretnénk elindítani alkalmazásokat. Hasznos segédprogram lehet egy alkalmazások indítására alkalmas program, amely tárolja a korábban futtatott programok nevét, majd a következő indításnál már egy listából is kiválaszthatjuk a megfelelő elemet. A kész program képe az *1. ábrán* látható.

Mielőtt egy programot elkezdünk fejleszteni, mindenképpen érdemes néhány mondatban megfogalmazni, hogyan kell annak működnie. Jelen esetben célunk egy olyan program elkészítése, mely megkönnyíti egy egyszerű ablakkezelő futtatása mellett más programok indítását. A programot saját magamnak készítettem, az ***IceWM*** ablakkezelő ***„Futtatás...”*** menüpontjának megvalósítására. Ebben az ablakkezelőben a programok indítására alkalmazható bármilyen program, melyet a konfigurációs állományban megadunk az ablakkezelőnek. Elsődleges célom, hogy a program könnyen érthető legyen és emellett megvalósítsa a kitűzött célt.

A példában szereplő program kétféle fő funkciót valósít meg. Elsősorban az alsó mezőben megadott programot fogja elindítani, majd amennyiben az nem szerepel a felső listában, kilépés előtt tárolja az új listaelemmel kibővített programlistát. Ha nem írunk be semmilyen új programot a beviteli mezőbe, akkor a listából kiválasztott alkalmazást indíthatjuk el. A program konfigurációs állománya a felhasználó saját könyvtárában lévő ***.flrun.cfg*** állomány. Ebben az állományban minden sor egy-egy futtatható programot ír le. A példában szereplő program indulásakor beolvassa ezt az állományt, majd feltölti a listát a beolvasott sorokkal. A felhasználó ezután beírhat új elemet vagy választhat a már meglévők közül. A program felhasználói felületén mindössze három gomb segíti a használatot. Az alapértelmezett (ENTER billentyűvel működtethető) gomb indítja a kiválasztott vagy megadott alkalmazást. A második gomb az alkalmazásból való kilépésre szolgál és a főablak bal alsó sarkában kapott helyet. Amennyiben a konfigurációs állomány alapállapotba állítására van szükség, ezt a műveletet a középső gomb (***„Reset config”*** felirattal) segítségével végezhetjük el. Természetesen a ***Linuxban*** megszokott módon, tetszőleges szövegszerkesztő programmal is megváltoztathatjuk a beállításokat tároló állományt.



1. ábra Az flRun alkalmazás ablaka

A program készítése során fontos szempont volt számomra, hogy bemutassam az ***eFltk*** grafikus fejlesztőeszközének használatát, így nem használtam más szövegszerkesztőt vagy fejlesztői környezetet a megvalósítás során. Mindössze az ***eFluid***-ra és egy terminál ablakra volt szükségem, melyben a fordítást időnként elvégeztem, majd a programot elindítottam.

Lássunk tehát hozzá a megvalósítás lépéseinek. Miután kialakítottuk a felhasználói felület képét, hozzunk létre egy új függvényt. Ez az első lépés szükséges ahhoz, hogy a főprogram ablakát megtervezhessük és elhelyezhessük rajta a megfelelő elemeket. Tehát a fejlesztőkörnyezet menüjében keressük meg a ***„New”*** menüt és válasszuk ki a ***Code->Function/Method*** menüpontokat. A megjelenő párbeszédablakban hagyhatunk mindent az alapértelmezett beállításon. Ezután hozzunk létre egy új ablakot a ***New->Group->Window*** pontok választásával. Állítsuk be a méretét, majd kattintsunk rá kettős kattintással. Itt megváltoztathatjuk például az ablak háttérszínét vagy címét, de leginkább a ***„C++”*** fülecskén lévő adatok megadása fontosabb.

Az ablak neve ebben a példában `MainWindow` és az ablak létrehozásakor még végrehajtandó kódrészletet az ***„Extra code”*** mezőben kell begépelni. A kódrészlet az *1. listában* látható. Ez a kód fogja lekérdezni a felhasználó saját könyvtárát, majd azt kiegészíti a konfigurációs állomány nevével. Ez a változó a ***homedir***, mely a program futása során mindvégig használható.

A hozzáadott kód utolsó sorában még a `load_config()` függvény segítségével betöltjük a konfigurációs állományt és feltöltjük a beolvasott elemekkel a legördülő listát.

A függvény sorai a *2. listán* tanulmányozhatók a további kiegészítő függvények mellett. A programban mindössze négy különleges szerepű eljárásra van szükség.

1. lista

```
homedir=(char *)calloc(128, sizeof(char));
homedir=getenv("HOME");
strcat(homedir, CONFIGNAME);
load_config(homedir);
```

Az `initialize()` függvény a program indulásakor memóriát foglal a listaelemek számára. A `cleanup()` függvény tulajdonképpen ennek a memóriaterületnek a felszabadítására szolgál továbbá a program befejeződése előtt felszabadítja a `homedir` változó számára foglalt memóriát is. Mivel a programból többféleképpen is ki lehet lépni, helyesnek láttam a memória felszabadítására szolgáló kódrészeket külön függvényben megvalósítani.

A listában látható másik két függvény a konfigurációs állomány betöltésére és mentésére szolgál. A mentésre csak abban az esetben van szükség, ha a listában még nem szerepel a felhasználó által megadott program neve.

A lista elején még néhány állandót határozunk meg, melyek megkönnyítik a program további bővítését. Kezdetben megelégedtem maximálisan tíz listaelem tárolásával, hiszen ezzel a programmal általában a gyakran használt alkalmazásokat fogom futtatni.

A legördülő lista megvalósítására az `Fl_Choice` objektumot használjuk majd, a kézi adatbevitelhez pedig az `Fl_Input` vezérlőelemet.

Miután elhelyeztük a vezérlőelemeket az ablak területén, ideje lesz a működést is megvalósítani a különféle ***visszahívó (callback)*** függvények segítségével. Mindössze a három gombra kell meghatározni a függvényeket. Az egyes elemekre duplán kattintva előbukkan a beállításokat segítő párbeszédablak, melyben minden elemnek érdemes egy beszédes nevet adni. A visszahívó függvények megvalósítására is ebben a párbeszédablakban van lehetőségünk.

Kezdjük tehát a konfigurációs állomány alapállapotba hozásával. A függvény rendkívül egyszerű, mindössze egy kérdező ablakban (`fl_ask()` függvény) megerősítést kérünk a felhasználótól, majd annak pozitív válasza után töröljük a konfigurációs állományt és a listaelemeket. Az állomány törléséből nem származhat semmilyen hátrányunk, hiszen amennyiben a program indulásakor nem található ilyen állomány, a program folytatja futását és üres listából választhatunk. A felhasználó által megadott alkalmazást kilépéskor a beállításokat tároló állományba írjuk, így újra létrehozzuk és tároljuk a beállításokat. Az állomány törlésére az `unlink()` rutint használjuk, tehát a programunk elején szükséges az `#include <unistd.h>` sor megadása. A listaelemek törlésére az `Fl_Choice` objektum `remove()` függvényét használjuk egy ciklusban végrehajtva.

A megadott vagy kiválasztott alkalmazás indítására a ***„Run program"*** gomb visszahívó függvényében lesz lehetőségünk. A függvény egy lehetséges megvalósítását a *3. lista* mutatja.

A függvényben elsősorban megvizsgáljuk, hogy kézi adatbevitel történt-e vagy a felhasználó a listából választott valamilyen programot. Az új alkalmazás megadásakor

2. lista

```
#define MAX_ELEMS 10
#define CONFIGNAME "/.flrun.cfg"
static char *config[MAX_ELEMS];
static char *homedir;

void save_config() {
 int i;
 FILE *ofile;
 ofile=fopen(homedir, "w+");
 if (!ofile) {
   fprintf(stderr, "cannot open output file\n");
   ➥return;
 }
 for (i=0; i<proglist->size(); i++) {
   if (proglist->size()) {
      Fl_String str=proglist->text(i);
      fprintf(ofile, "%s\n", (char*)str);
   }
 }
 fclose(ofile);
 return;
}

void load_config(const char* confname) {
 int i;
 FILE *infile;
 if (confname==NULL) {
   fprintf(stderr,
      "config filename is invalid\n");
 return;
 }
 infile=fopen(confname,"r");
 if (!infile) {
 fprintf(stderr, "configfile not found\n");
 ➥return;
 }
 for (i=0; i<MAX_ELEMS-1; i++) {
 if (feof(infile)) break;
 fscanf(infile, "%s\n", config[i]);
 if (config[i]) proglist->add(config[i]);
 }
 fclose(infile);
 return;
}

void initialize() {
 int i;
 for (i=0; i<MAX_ELEMS-1; i++)
   config[i]=(char*)calloc(128, sizeof(char));
}

void cleanup() {
 int i;
 for (i=0; i<MAX_ELEMS-1; i++)
   if (config[i]) free(config[i]);
 free(homedir);
}
```

3. lista

```
static void cb_btOK(Fl_Return_Button*, void*) {
 char *mstr=(char *)calloc(128, sizeof(char));
 strcpy(mstr, proginput->value());
 strcat(mstr, (const char*)"&");

 // Nem adtunk meg kézzel semmit
 if (strncmp(mstr, "&", 1)) {
   system(mstr);
   free(mstr);
   mainWindow->hide();
   // Ha meg nincs benne a listaban a program,
   // akkor fel kell vennünk
   if (proglist->size()<10) {
      if (!proglist->find(proginput->value())) {
         proglist->add(proginput->value());
         // mentjuk a listat
         save_config();
      }
   }
   return;
 }
 if (proglist->size()>0) {
   Fl_String str=proglist->text(proglist->
   ➥value())+"&";
   if (!str) return;
   system(str.c_str());
 }
 mainWindow->hide();
}
```

a `system()` hívással végrehajtjuk a megadott psrogramot és megvizsgáljuk a listaelemeket. Ha nem találjuk a legördülő lista elemei között az alkalmazás nevét, hozzáadjuk a meglévő elemekhez és a `save_config()` függvény meghívásával tároljuk a beállításokat.
Ha a listából választott a felhasználó, akkor nincs más tennivaló, mint kiolvasni az elem szövegét és szintén a `system()` hívással végrehajtani a programot. Mindkét esetben az ablakot elrejtjük a `hide()` metódusának segítségével, hiszen az ***eFltk*** fő ciklusa addig fut, amíg valamilyen programablak látható a képernyőn. A program tehát befejezi működését a gomb megnyomása után.
A harmadik visszahívó függvényben egészen egyszerűen csak elrejtjük a fő ablakot, így a ***„Cancel"*** gomb használata szintén a program befejeződését eredményezi.
Érdemes még néhány szót ejteni a főprogramról is. Az ***eFluid***-ban a ***New->Code->function/method*** menüpontok választásával hozhatjuk létre a `main()` függvényt. A megjelenő párbeszédablakban minden beviteli mező tartalmát töröljük ki, hogy az ***eFluid*** tudtára adjuk, hogy a `main()` függvény kódját szeretnénk megadni. A főprogram kódja mindössze a 4. listában látható néhány sorból áll.

4. lista

```
int main (int argc, char **argv) {

 initialize();
 mainWindow=make_window();
 mainWindow->show();
 Fl::run();
 cleanup();
 return 0;
}
```

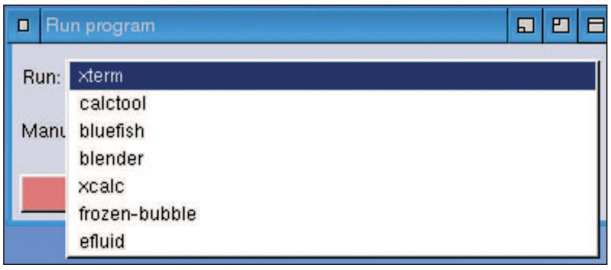
A függvényben meghívjuk a memóriaterületek lefoglalására szolgáló `initialize()` függvényt, majd létrehozzuk és megjelenítjük a program ablakát. Ezután már a szokásos `Fl::run()` függvény, vagyis az eseménykezelő ciklus futtatása következik. A ciklus befejeződése egyben az alkalmazás futásának végét is jelenti, azonban a kilépés előtt a `cleanup()` függvény segítségével felszabadítjuk a lefoglalt memóriaterületeket.
A fenti függvények megvalósítása után már nincs más tennivalónk, mint menteni a grafikus felületet és vele együtt a programot is. Az ***eFluid***-dal készült alkalmazások kiterjesztését érdemes ***.fl***-ként megadni, mert a fejlesztőeszköz is ilyen kiterjesztésű állományokat tekint sajátjának és alapállapotban ezeket jeleníti meg az állományok megnyitását segítő párbeszédablakban is.
A mentést később billentyűk segítségével is elvégezhetjük, mégpedig a ***CTRL+S*** kombináció alkalmazásával.
Ne feledkezzünk meg a forrásállományok elkészítéséről sem. Használjuk a ***CTRL+W*** billentyűket, így az ***eFluid*** elkészíti a teljes programunkat tartalmazó forráskódot és hozzáfoghatunk a program fordításához. Ennek legegyszerűbb módja, ha egy terminál ablakban kiadjuk a következő parancsot:

```
efltk-config ---compile runapp.cpp
```

Ebben az esetben a grafikus felületet ***runapp.fl*** néven mentettem és így a generált programkódot tartalmazó állomány neve is megegyezik a mentés során megadottakkal. A parancs hatására létrejön a futtatható állomány, amit érdemes megszabadítani a felesleges dolgoktól.
Adjuk ki a

```
strip -s ./runapp
```

parancsot - a program könyvtárában -, hogy a futtatható állományt megszabadítsuk a szerkesztés során létrejött szimbólumoktól. A program futását ez a művelet nem befolyásolja, viszont jelentősen lecsökkenthetjük a futtatható állomány méretét.
Amennyiben összetett programrendszert készítünk érdemes megismerkedni a kézi fordítás mikéntjével is. Amikor az ***eFltk*** segédprogramját használjuk látható, hogy milyen kapcsolókra van szükség a fordításhoz. Ezeket a kapcsolókat kiegészíthetjük tetszőleges ***GCC*** opciókkal vagy készíthetünk egyszerű ***Makefile***-t is a fordítás megkönnyítéséhez.



2. ábra Alkalmazás kiválasztása

Ebben a példában néhány kiegészítő opciót is megadtam a fordítónak, majd az alábbi parancssor segítségével, kézzel fordítottam le a programot:

```
gcc -o runapp -O3 -funroll-loops -fexceptions
➥-march=pentiumpro -DFL_SHARED
    -I/usr/X11R6/include -L/usr/X11R6/lib -lefltk
    ➥-lX11 -lXext -lm runapp.cpp
```

A program elkészítése után ne felejtsünk el biztonsági másolatot létrehozni az állományokról.
Végül nem maradt más hátra, mint az *IceWM* tudtára adni, hogy mostantól programok futtatására ezt a programot szeretnénk használni. Nyissuk meg az *IceWM* beállításait tartalmazó *~/.icewm/preferences* állományt (amennyiben nem találunk ilyen állományt, az *IceWM* adatait tartalmazó könyvtárból át tudjuk másolni az alapértelmezett beállításokat), és keressük meg benne a `RunCommand=` kezdetű sort. Itt kell megadnunk a futtatható állomány elérési útját, majd az *IceWM* újraindítása után máris használatba vehetjük ezt a hasznos segédprogramot. A *2. ábrán* látható az elkészült program működés közben.
Ezzel végére ért az *eFltk* bemutatását célzó írások sora. Természetesen nem adhatok az adott keretek között teljes referenciát minden osztályról, de remélhetőleg sikerült ízelítőt adnom a grafikus elemkészlet lehetőségeiből és a vállalkozó kedvű, tanulni vágyó olvasók folytatják a készlet felfedezését és használatát. Ehhez nagy segítség lehet a továbbiakban az *Fltk* dokumentációja, hiszen nem minden objektumról találunk leírást az *eFltk*-ban. Az eredeti *Fltk* dokumentáció azonban részletesen, példákkal megvilágítva bemutatja az elemkészlet minden lehetőségét, legyen szó szövegszerkesztő készítéséről vagy *OpenGL* alapú három dimenziós megjelenítésről.
A készlethez az interneten is találhatunk kiegészítő vezérlőket és számos példán keresztül elmélyedhetünk használatának rejtelmeiben. Izgalmas felfedező utat kívánva egy kis időre búcsúzom kedves olvasóimtól, jó nyaralást és tanulást kívánok mindenkinek.

Fábián Zoltán (dzooli@freemail.hu)
26 éves, jelenleg oktatóként dolgozik, szabadidejében szívesen foglalkozik Blenderrel, programozással és elektronikai tervezéssel. Szereti a természetet, túrázást, úszást és a kellemes baráti társaságot.

Adatbázis-kezelés PHP-ben, PEAR módra

Építkezni fogunk, mégpedig PEAR modulokból. Az első górcső alá vett – és egyben legfontosabb – építőkő az adatbázis-kezelés kategóriát megalapozó DB csomag.

Sorozatunk előző epizódjában áttekintettük a PEAR rendszer alapjait, filozófiáját, megismerkedtünk az alapprogram telepítésével, majd egy egyszerű példán keresztül megismerkedhettünk egy előre elkészített modul telepítésével és használatával. A továbbiakban elsőként a méltán népszerű *PEAR DB* csomagot szeretném bemutatni, amely az egyik legöregebb, ennél fogva legjobban kiforrott elem, számos más adatbázis-kezelést végző modul épül rá, és igen jelentős felhasználói táborral rendelkezik.

A *DB* csomag fogalom a *PEAR*-t használó fejlesztők körében. Egy olyan adatbázis absztrakciós rétegről van szó, amely áttetszővé teszi a modult használók számára a rendszer alatt futó adatbázist: nem kell tudni, hogy milyen kiszolgáló fut a gépen, nem kell ismerni a sajátosságait, és adott esetben még az adatbázis-kezelőt is ki lehet cserélni a rendszer alatt, bár ez utóbbi nem szokás. A választott adatbázis-kezelő a tervezés elején meghozott stratégiai döntés, nem célszerű megváltoztatni, mert a következmények nem ismertek pontosan. Annyiból viszont rendkívül hasznos, hogy módosítás nélkül álljunk át az adott adatbázis-kiszolgáló egy újabb, némileg eltérő változatára.

A működés lényege, hogy az adatbázisunkat nem a megszokott módon, a *PHP* beépített függvényeivel kezeljük, hanem a *DB* osztály metódusainak hívásával, így fedve el előlünk a használt adatbázis-kezelőt. Ezek a metódusok egyrészt jópár feladatot átvesznek a fejlesztőtől, másrészt segítik a különböző alkalmazások, projektek egységesen történő felépítését.

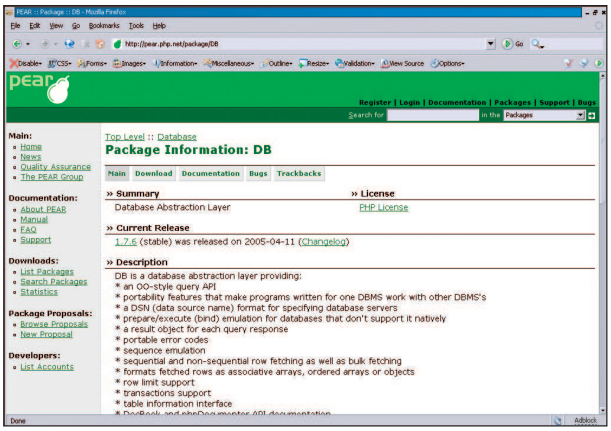
Lássunk egy konkrét példát!

Egy *mysql* adatbázisból szeretnénk egy egyszerű lekérdezés adatait kinyerni. Ehhez a következőt kell tennünk a *PHP* beépített függvényeit használva:

```
$dbh =
mysql_connect('kiszolgalo','felhasznalo','jelszo');
mysql_select_db('adatbazis_nev');
$res = mysql_query('select * from test',$dbh);
//az eredmények bejárása...
```

Mindez PEAR DB-n keresztül az alábbi módon történik:

```
$dbh =&
```



1. ábra A PEAR DB csomag weboldala

2. ábra A végfelhasználói dokumentáció példaprogramokkal

```
DB::connect("mysql://felhasznalo:jelszo@kiszolgalo/
➥adatbazis_nev");
$res =& $dbh->query("select * from test");
//az eredmenyek bejarasa...
```

(Megjegyzés: a referencia átadást végrehajtó operátorra (=&) csak *PHP4* esetében van szükség, *PHP5*-ben minden objektum referencia formájában adódik át az egyenlőség operátor hatására.)

Mint látható, minden későbbi művelet során a kapcsolatkezelő (`$dbh`) metódusait kell meghívnia *PHP* függvények helyett, ezzel egy egységes adatbázis-kezelő felületet kaptunk, amit mindenhonnan, mindenféle adatbázis-kiszolgáló mellett ugyanúgy kell használni.
Mielőtt részletes bemutató formájában megnéznénk, hogy hogyan is kell pontosan használni, tekintsük át a megoldás felépítését. Maga a csomag több osztályból áll, ennek leggyakrabban használt eleme a *DB* osztály maga, ez képviseli a felhasználók felé a *PEAR DB*-t, ezen az osztályon keresztül lehet csatlakozni, kapcsolatot bontani, lekérdezni, hibát kezelni, stb. Amikor egy kapcsolat felépül, a használat adatbázistól függően más-más módon kell az elvégzendő műveleteket leképezni. Erre szolgálnak a közvetlen adatbázis osztályok, amelyekből minden adatbázistípushoz (*MySQL*, *PostgreSQL*, *Oracle*, stb) pontosan egy tartozik. A tényleges műveletek tehát egy ilyenen keresztül zajlanak, s az értékek továbbadódnak a *DB* osztálynak, amelyik visszaadja a felhasználók felé. Ezek az osztályok a *DB* felé egyformák, mindet ugyanúgy kezeli a központi osztály, miután a kapcsolat felépítése során kiválasztotta a neki megfelelőt. Az egyformaságot úgy érik el, hogy minden ilyen adatbázis-függő közvetlen osztály egy azonos ősből, a `DB_common` alaposztályból származik, ezáltal örökli annak tagfüggvényeit. A már emlegetett *adatbázis >> függő közvetlen osztály >> DB osztály >> felhasználó* lánc legvégén (a felhasználó előtt) még van egy elem, ugyanis a felhasználó egy eredményhalmazt kap, amely itt az egyértelműség kedvéért szintén objektum, amely magában hordozza az adatok barátságos kinyeréséhez szükséges metódusokat. A visszaadott eredményhalmaz a `DBresult` *burkoló (wrapper) osztály* egy példánya.
Ezen kívül van még pár osztály, amely az alapok elsajátítása szempontjából nem lényeges, ezekre most nem térünk ki, inkább lássuk magát a medvét! Itt is meg kell jegyeznem, hogy nincs lehetőségünk az összes tagfüggvény áttekintésére, így arra fogunk törekedni, hogy egy teljes, jól áttekinthető „fejlesztői” folyamatot kapjunk kézhez a cikk végén.

Első lépés: kapcsolódás az adatbázishoz

Megszokhattuk, hogy ahányféle adatbázis, annyiféle kapcsolódási mód. A *DB* esetében ez azonban egységesítve van. Megfigyelhetjük használat közben is a fenti példakód alapján: a csatlakozáshoz egy *URL*-t definiál *(DSN – Data Source Name ~ Adatforrás név)*, amellyel leírható az adatbázis típusa, helye, felhasználónév, jelszó és az adatbázis neve, és még sok egyéb más is. A leírás kétféleképp történhet: meghatározott formátumú karakterlánccal, vagy meghatározott elemeket tartalmazó tömbbel. Sajnos adatbázis típusonként eltérő lehet a megadható komponensek neve és száma, általánosságban csak annyit lehet elmondani, hogy a *DSN* az alábbiak szerint alakul:

```
phptype(dbsyntax)://username:password@protocol+host
➥spec:port/database?option=value
```

A magyarázat:

```
phptype: az adatbázis típusa, amelyet a php kezel
dbsyntax: adatbázis „altípus” pl. ODBC vezérlő esetén
username:password: név/jelszó páros
protocol: meghatározza, hogy milyen protokollon
keresztül érjük el az adatbázist (tcp, unix socket,
stb.)
hostspec:port: Az adatbázis-kiszolgáló neve és
portja
database: a használni kívánt adatbázis
option=value&option2=value2: kulcs érték párok,
amelyekkel speciális paramétereket adthatunk meg
a kapcsolat számára.
```

A fenti példában *MySQL* adatbázis felett, „root” felhasználóként, jelszó nélkül csatlakoztunk a helyi gépen futó kiszolgálóhoz, ahol a ‘test’ nevű adatbázist használtuk.
Ha tömb formájában akarjuk megadni az adatokat, akkor asszociatív módon, kulcs-érték párokkal tehetjük meg, ahol a kulcsok a fent részletezett elemek, értékük pedig a behelyettesíteni kívánt karaktersorozatok. A kapcsolódáshoz ezután a `DB` osztály `connect()` metódusát használhatjuk, melynek első paraméter a már emlegetett *DSN*, a második pedig a kapcsolat paramétereit rögzítő tömb. Ez tartalmazza például, hogy perzisztens legyen-e a kapcsolat, használjon-e *SSL* protokollt, a nyomkövetési üzenetek szintjét, stb.
A sok elmélet után következzen egy kis gyakorlat, nézzünk egy-egy példát a karakterlánc, illetve tömb formájában átadott kapcsolódási paraméterekkel.

```
$dsn = ‘pgsql://someuser:apasswd@localhost/
➥thedb’;
$options = array(
    ‘debug’       => 2,
    ‘portability’ => DB_PORTABILITY_ALL, //minden
eszközzel támogassa az AB-függetlenséget
);

$db =& DB::connect($dsn, $options);
if (PEAR::isError($db)) {
    die($db->getMessage());
}
```

A másik esetben a kód így néz ki:

```
$dsn = array(
    ‘phptype’  => ‘pgsql’,
    ‘username’ => ‘someuser’,
    ‘password’ => ‘apasswd’,
    ‘hostspec’ => ‘localhost’,
    ‘database’ => ‘thedb’,
);
$options = array(
    ‘debug’       => 2,
    ‘portability’ => DB_PORTABILITY_ALL, //minden
eszközzel támogassa az AB-függetlenséget
);

$db =& DB::connect($dsn, $options);
if (PEAR::isError($db)) {
    die($db->getMessage());
}
```

Bizonyos esetekben néhány extra paraméter is belekerül a tömbbe, ilyen a fájl alapú ***SQLite*** adatbázis-kezelő, ahol a `mode` paraméterrel adható meg a fájl oktális jogosultsága (például `644`). Ezekről a ***DB*** csomag leírásában tájékozódhatunk.

Második lépés: a lekérdezés indítása

Ha ezzel megvolnánk, sikerült kapcsolatot teremteni a kiszolgálóval, jöhet az adatok lekérése. Ez alapesetben rendkívül egyszerű, a `DB` osztály `query()` tagfüggvényének segítségével paraméterként átadjuk az ***SQL*** lekérdezést tartalmazó karakterláncot. Az eredményt egy `DBresult` típusú objektum képében kapjuk meg.

```
include("DB.php");

$db = DB::connect("mysql://root@localhost/test");
$res = $db->query("select * from proba");
```

A dolog némiképp bonyolódik, ha szeretnénk paraméteres lekérdezéseket készíteni. Ebben az esetben a lekérdezési karakterláncban nem szerepelnek konkrét értékek, hanem azt a lekérdezőfüggvény egy másik paramétereként adjuk át.

```
$res = $db->query("select * from proba where ertek
➥= ?", 26);
```

Ekkor a kérdőjel helyére a 26-os szám helyettesítődik be. Több paraméter esetén egy tömbként kell átadni az értékeket, ahol a tömb elemei az egyes paramétereket jelentik sorrendben balról jobbra.

```
$res = $db->query("select * from proba where kulcs
➥= ? and ertek = ?", array(13,26));
```

Harmadik lépés: az eredmények kezelése

A lekérdezést legtöbb esetben azért adjuk ki, hogy feldolgozzuk annak eredményét. Szó volt róla, hogy az eredményt egy `DBresult` típusú objektum képében kapjuk meg, amely magában hordozza a feldolgozáshoz, adatkinyeréshez szükséges tagfüggvényeket.
Menjünk végig egy eredményhalmazon, és írassuk ki az egyes sorokat:

```
include("DB.php");
$db = DB::connect("mysql://root@localhost/test");
$res = $db->query("select * from proba");
while ($res->fetchInto($row, DB_FETCHMODE_ASSOC)) {
    var_dump($row);
}
```

A `fetchInto()` metódus második paramétere azt határozza meg, hogy milyen formában adja vissza az adathalmaz következő sorát. Alapértelmezetten a `DB_FETCHMODE_ORDERED` konstans van érvényben, amely egy számokkal indexelt tömbben tartalmazza sorban az egyes mezők tartalmát. A kódrészletben is alkalmazott `DB_FETCHMODE_ASSOC` asszociatív tömbbel tér vissza, ahol a kulcsok a mező nevei, az értékek a mező értékei. A harmadik lehetőség a `DB_FETCHMODE_OBJECT` konstans használata, amelynek hatására a visszatérési érték egy objektum lesz, ahol az osztályváltozók neve a mezőnevekkel egyezik meg, az értéke pedig a mezők tartalmával.
Az eredményhalmazból egyéb értékes információ is kideríthető. A `numRows()` metódus visszaadja, hogy hány sorral tért vissza a lekérdezés, a `numCols()` ehhez hasonlóan azt, hogy hány oszlopa van az eredményhalmaznak. Ezen kívül lekérdezhetjük az halmaz részletes felépítését a `tableInfo()` tagfüggvény segítségével, amely tartalmazza, hogy milyen oszlopai vannak az eredménynek, és azok milyen típusúak, milyen méretűek.

Negyedik lépés: lekérdezés indítása előkészítéssel

Számtalan esetben előállhat az a helyzet, hogy hasonló módosító jellegű lekérdezéseket kell futtatnunk, ahol kizárólag a módosított értékek változnak, a lekérdezés többi része ugyanaz marad. Erre számtalan adatbázis-kezelő alkalmazza azt a megoldást, hogy paraméteres lekérdezéseket használva első lépésben előkészíti, „lefordítja" a lekérdezést, majd ezt az előkészített parancsot tudjuk később, akár egymás után sokszor lefuttatni.
Például nézzük azt az esetet, amikor minden alkalmazott fizetését meg kell növelni egy előre meghatározott összeggel, amit a vezetőség állított elő személyre szólóan. Az ***SQL*** parancs az alábbiak szerint néz ki:

```
'update employees set salary=salary + ? where
➥id = ?'
```

A fentiekből már tudjuk, hogy a lekérdezés indítása során helyettesíthetjük be a ? jelek helyére a kívánt értékeket. A lekérdezést előkészítve, majd az előkészített parancsot mindig a megfelelő paraméterrel elindítva egyetlen fordítással megoldható a művelet.
A ***PEAR DB*** biztosítja számunkra ezt a lehetőséget. A példát lefordítva az alábbi kódrészletet kell végrehajtanunk:

```
include("DB.php");
$db = DB::connect("mysql://root@localhost/test");
$qry='update employees set salary=salary + ? where
➥id = ?';
$sth=$db->prepare($qry);
foreach ($salary_lift as $id=>$value) {
        $db->execute($sth,array($id,$value));
}
```

Jól látható, hogy egy referenciát kapunk vissza az előkészített lekérdezésre, amelyet aztán ugyanúgy kell elindítani, mintha lekérdezés lenne, még a paramétereket is ugyanabban a tömbös formában kell átadni, ha több paramétert szeretnénk használni. Szót kell ejtenünk a lekérdezésben használatos helyfenntartókat (placeholder), ami az eddigiekben csak a ? jel volt. Ezeknek a helyére helyettesítődnek be a második paraméterként átadott változók.
PEAR DB terminológiával a ? jel skalár értékek helyét jelöli. A behelyettesítés során automatikusan átalakítja a speciális karaktereket (escape), illetve ha karaktersorozatról van szó, aposztrofok közé teszi a ***DB*** a használt adabázis-kezelő elvárásainak megfelelően.

A ! jel működése megegyezik az előzővel, azt leszámítva, hogy ott az értékek úgy helyettesítődnek be, ahogy átadtuk, tehát nincs speciális karakterek átalakítása és aposztrofok közé tétel.
Az utolsó az & jel, amely egy valódi fájlnevet vár a behelyettesítési folyamat során. A behelyettesített érték ekkor a megadott fájl tartalma lesz. Ez olyan esetekben használható jól, ahol fájlokat, vagy más bináris adatokat szeretnénk az adatbázisba tölteni.
Felmerülhet sokunkban a kérdés a fenti példa alapján: ha egyszer úgyis tömbben vannak az adatok, és a megoldást pont arra találták ki, hogy ismétlődő szerkezetű adatokat minél gyorsabban lehessen adatbázisba tenni, akkor mi a fenének bejárni a tömböt, és egyenként betenni az értékeket? A válasz az, hogy nem kell, létezik erre is megoldás, amelyet az `executeMultiple()` tagfüggvény valósít meg.

```
include("DB.php");
$db = DB::connect("mysql://root@localhost/test");
$qry='update employees set salary=salary + ? where
➥id = ?';
$sth=$db->prepare($qry);
$db->execute($sth,$salary_lift);
```

Az előkészítés-indítás jelleggel alkotott lekérdezést nem támogatja minden adatbázis-kezelő, de mi bárhol használhatjuk, mivel a `DB` alapértelmezetten emulálja ezeket a funkciókat számunkra (némi sebességcsökkenés árán természetesen).

Ötödik lépés: hibakezelés

A PEAR statikus függvényt kínál az eredményhalmazok, adatbázis leírók ellenőrzésére. Ha valamely művelet során hiba történt, akkor az adott művelet visszatérési értéke egy `DB_Error` típusú objektum, amely ugyanazokat a tagfüggvényeket kínálja, mint a `PEAR_Error` osztály. Ha tehát hibát szeretnénk észlelni, akkor a hiba elkövetése után a művelet során keletkezett eredményhalmazt kell vizsgálat tárgyává tenni, mégpedig a `PEAR` (ill. `DB_Error`) osztály statikus `isError()` tagfüggvényének segítségével.

Kapcsolódási hiba észlelése

```
$db = DB::connect("mysql://root@localhost/test");
if (PEAR::isError($db)) {
        die('Kapcsolódás sikertelen');
}
```

Lekérdezés futtatási hiba észlelése

```
include("DB.php");
$db = DB::connect("mysql://root@localhost/test");
$qry='update employees set salary=salary + ? where
➥id = ?';
$sth=$db->prepare($qry);
$res=$db->execute($sth,$salary_lift);
if (PEAR::isError($res)) {
        $res->getMessage();     //visszaadja
                                //a hibaüzenete
        $res->getCode();        //visszaadja
                                //a hibakódot
}
```

Mint látható, a `DB_Error` osztály is magában hordozza a hibakezelésre vonatkozó összes tagfüggvényt.

A PEAR DB előnyei

A fentiek alapján nem kell sokat ecsetelni a megoldás előnyeit: egyszerű, kis túlzással szabványos is, széles körben elterjedt, támogatott, dokumentált és nem nekünk kell megírni. Ez utóbbi két szempont igen jelentős: nem csak a kód megírására fordítandó időt takarítjuk meg, de a programhibák száma is minimális. Ezen túl már említettem, hogy ha más fejlesztő nyúl a kódhoz, nagy valószínűséggel ismerni fogja. Ha azonban ez nincs is így, a csomag igen jól dokumentált: nem csak ***API*** referencia áll rendelkezésre, de a ***PEAR*** honlapján részletes végfelhasználói dokumentációt találunk szemléletes példákkal illusztrálva.
A ***DB*** megoldásai rendkívül hasonlítanak például a ***JDBC*** filozófiához, ahol szintén minden adatbázist ugyanúgy kell kezelni: objektum-orientált és átgondoltságot sugároz, így nem csak ***PEAR DB*** fejlesztők tudják hatékonyan elsajátítani a használat rejtelmeit, de olyanok is, akik ***JDBC*** adatbázis absztrakciós rétegen nőttek fel.

A PEAR DB hátrányai

A modul rendkívül általános, számtalan dolgot tud, ennélfogva igen valószínű, hogy számtalan szolgáltatását nem is fogjuk soha igénybe venni. Ezek csak ott vannak, lassítják a kódot, stb. Másfelől a valódi funkciók elrejtése (előkészítéses lekérdezés nem támogatott adatbázison, eredményhalmazok burkolása, stb). jelentős többletterhelést okoz a programnak. Ez általában igaz minden emelt szintű felületre is. A dolog legteteje pedig az, hogy a kódunk hordozható marad az adatbázis-kiszolgáló fölött, annak típusától függetlenül. Az elején már említettem, hogy ezt a lehetőséget a legritkább esetben alkalmazzuk. Igaz, lehet a kompatibilitás mértékét szabályozni a kapcsolat felépítésekor, de még így is számtalan felesleges menet közbeni átalakítást cipel a hátán a rendszer.
Mindezek ellenére személy szerint egyértelműen javaslom a használatát! A hétköznapi alkalmazások esetében soha nincs olyan teljesítményigény, amely kizárttá tenné a ***DB*** alkalmazásának lehetőségét, viszont szép, áttekinthető, átgondolt felépítésű kóddal büszkélkedhetünk.
A ***DB*** csomagról igen részletes leírást talál az érdeklődő olvasó a ***http://www.pear.php.net/manual/en/package.database.db.php*** címen. Ez a végfelhasználói dokumentáció sok-sok kódrészlettel.
Aki ezzel nem elégszik meg, és ki akarja vesézni a teljes csomagot, annak ajánlom a ***http://www.pear.php.net/package/DB/docs/latest/*** címen elérhető ***API*** referencia leírást. Kevésbé érthető, mint az előző, viszont tényleg minden aprólékos információ megtalálható benne.

Komáromi Zoltán

KAPCSOLÓDÓ CÍMEK

A PEAR DB elérhetősége:
➲ http://www.pear.php.net/package/DB/

Kávéfőzés lépésről lépésre (3. rész)
Tömbök és interfészek, avagy a változatosság gyönyörködtet

Az előző részben megismerkedtünk az öröklődés fogalmával. Ennek a módszernek a segítségével egy meglévő osztályt bővíthetünk újabb tulajdonságokkal és műveletekkel. Ehhez gondos tervezés esetén nincs szükség az ősosztály módosítására, valóban nincs másról szó, mint egyszerű kiterjesztésről.

Példánkban egy állatokat jelképező osztályból örököltettünk egy `Tehen` nevű osztályt, amely új tulajdonságát, a naponta termelt tej mennyiségét ki is tudta íratni a képernyőre.
Ahhoz, hogy ennél a kiíratásnál megjelenítsük az állat hangját is, elveinknek ellentmondva módosítást hajtottunk végre az ősosztályban. Az `Allat` osztály `hang` változójának láthatóságát személyesről átállítottuk védettre. Erre nem lett volna szükség, ha tudjuk előre, hogy öröklődést szeretnénk majd alkalmazni. Ez tehát elkerülhető egy átgondolt modell esetén, aminek megalkotásában nagy segítségünkre lehetnek az ***UML*** diagramok, melyek közül egyre szintén láttunk példát.
Vessünk most egy pillantást az elmúlt alkalommal készített alkalmazás belépési pontjára:

```
public static void main(String[] args) {
   Tehen tehen = new Tehen("múúú", 20);
   Allat lo = new Allat("nyihaha");
   Allat kutya = new Allat("vauvau");
   tehen.szolaljMeg();
   tehen.mennyiTej();
   lo.szolaljMeg();
   kutya.szolaljMeg();
}
```

Noha a forráskód az objektumközpontú filozófia számos ismertetőjegyét magán viseli, még mindig bőbeszédűnek érezhetjük a leírtakat. Felmerül a kérdés, hogy ha a `tehen`, a `lo` és a `kutya` objektumot is megszólaltatjuk, miért kell ezt három, független metódushívással megtenni. Gondoljunk bele, ha nem csak három objektumunk lenne, hanem mondjuk száz. Ez esetben százszor kellene leírni gyakorlatilag ugyanazt? Szerencsére nem.
Mindhárom objektumban van valami közös, mégpedig az, hogy az `Allat` osztályból származnak. A `lo` és a `kutya` közvetlenül, hiszen ezek az `Allat` példányai. A `tehen` ugyan a `Tehen` osztály példánya, de ez az `Allat` kiterjesztése, így rá is igaz az állítás. Ez a közös vonás pedig egyszerűen megragadható úgy, hogy objektumainkat egy tömbbe gyűjtjük össze.

Tömböm, tömböm mondd meg nékem

Java-ban a tömb szerkezet is objektumként jelenik meg. Ez azt jelenti, hogy egy tömb létrehozásához ugyanúgy a `new` kulcsszót kell használnunk mint eddig. Fontos különbség viszont, hogy a referencia változónk, mellyel a létrejövő objektumra tudunk hivatkozni valamilyen tömb típusú, és ezért egyben indexelhető is. Lássuk, hogyan néz ki az átírt `main()` függvény:

```
public static void main(String[] args) {
   Tehen tehen = new Tehen("múúú", 20);
   Allat lo = new Allat("nyihaha");
   Allat kutya = new Allat("vauvau");

   Allat[] allatok = new Allat[3];
   allatok[0] = tehen; allatok[1] = lo;
   ➥allatok[2] = kutya;

   for (int i = 0; i < allatok.length;
   ➥i ++) {
          allatok[i].szolaljMeg();
   }

   tehen.mennyiTej();
}
```

A létrehozott objektumokból egy `allatok` nevű tömböt készítünk, amint az a metódus 4. és 5. sorában látható. Az `allatok` létrehozása után újabb példányosítás már nem történik, csupán feltöltjük a tömböt az objektumok referenciáival. Így az eredeti néven túl egy másik úton

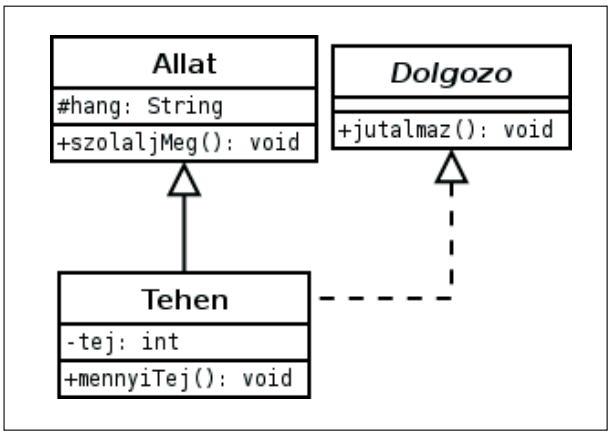
is megcímezhetjük ugyanazokat az objektumokat.
Az = operátorral csupán az objektumok elérhetőségeit másoltuk át a tömbbe, nem a teljes objektumokat.
Az ezután következő `for` ciklus *C*-ből ismerős lehet, viszont több ponton segít rajtunk a Java. A ciklus fejében három rész található, melyeket `;` választ el egymástól. Az első rész készíti elő a használandó változókat, mivel ez még a törzsbe való első belépés előtt lefut. Kényelmes és hasznos, hogy a ciklusváltozót elég itt létrehozni, és nem kell a teljes függvény legelején. Ennek az a haszna is megvan, hogy a ciklus lefutását követően ez a segédváltozó megszűnik. A második rész a törzs minden lefutása előtt kiértékelődik, és csak akkor folytatódik a ciklus, ha ez a feltétel teljesül. Itt ellenőrizzük, hogy a ciklusváltozó, amellyel a tömböt fogjuk indexelni, nem lépte-e túl a tömb méretét. Kihasználjuk, hogy a tömb egy objektum, és szerencsénkre a mérete egy tagváltozón keresztül lekérdezhető. Végül az utolsó rész az egyes lefutások után hajtódik végre, ezen a helyen növeljük a ciklusváltozót.
Így a ciklus háromszor fut le, előbb a tehenet, majd a lovat, végül a kutyát szólaltatja meg. Ezzel a módszerrel elértük, hogy ha növelni szeretnénk az állataink számát, a kiíratás részébe már nem kellene belenyúlnunk. Tetszőleges számú referenciát tartalmazhat az `allatok` tömb, az összes hivatkozott objektumnak meghívásra kerül a `szolaljMeg()` metódusa.
Jó kérdés, hogy ha ez ilyen remekül működik egy egyszerű metódus meghívásakor, miért ne oldhatnánk meg hasonlóan az objektumok létrehozását. Ekkor nem is lenne szükség másra, csak a tömbre, melynek elemei egy hasonló ciklusban egyenként kaphatnának egy új objektumot. Ennek jelen helyzetben több dolog is ellentmond. Egyrészt a tehén `Tehen` osztályból származik, és így noha hivatkozhatunk rá egy `Allat` típusú referenciával, de csak az `Allat` osztályban meglévő tagokat érhetjük el. Ha létrehoznánk egy `Tehen` típusú objektumot, de csak olyan referenciánk lenne rá, ami `Allat` típusú, elveszítenénk azt, amivel az ősosztályt kiegészítettük. Másrészt gond lenne a konstruktorok paraméterezésével is, hiszen honnan tudjuk egy ciklus belsejéből, hogy épp melyik hangot kell átadni. Ebben az esetben tehát ez nem egy jó ötlet, viszont több hasonló objektum létrehozásakor kényelmes megoldást jelenthet.

Díjazzuk a teheneket

Most játszunk el a gondolattal, hogy főnökünk hazaért az első vakációból és elmondja, milyen kiváló ötlete támadt, mialatt a Balaton partján pihent. Teheneink látható módon keményen dolgoznak, hiszen minden tehén objektumnak tulajdonsága a naponta termelt tej mennyisége. Jó lenne külön jutalommal díjazni ezt a megfeszített munkát. Ezért szükséges a jól dolgozó állatoknak egy `jutalmaz()` metódus, mellyel kifejezhetjük elismerésünket a teljesítményért.
Első ötletünk az lehetne, hogy egyszerűen vegyük fel a `Tehen` osztályban az említett eljárást, és mi is menjünk nyaralni. Emlékezzünk azonban vissza saját hibánkra, mikor egy egyszerű öröklöttetéshez bele kellett nyúlnunk az ősosztály kódjába. Ahhoz, hogy ne essünk újabb csapdába, gondoljuk végig, mit állítanánk azzal, ha jelenlegi modellünket úgy egészítenénk ki, hogy a `Tehen` osztályba minden további nélkül belevésnénk egy ilyen metódust. Ezzel azt fejeznénk ki, hogy a tehenek, és csak a tehenek jutalmazhatóak. Bármennyire is kedveljük ezeket a béke szimbólumaként ismert, szeretetreméltó állatokat, ezt nem jelenthetjük ki.
Ha már az objektumközpontúság a magas fokú kódújrahasznosítás kecsegtet, ne mondjunk ennek ellent egy helytelen modellel. A teheneket valóban meg lehet jutalmazni, ugyanakkor ez később igaz lehet a lovakra, vagy a kutyákra is, mi több, alkalmazásunk fejlődésével még emberekre is. Ezért az `Allat` osztály sem rendelkezhet egyértelműen a művelettel. Mit tehetünk ebben a helyzetben?
Elsőként gondolhatunk arra, hogy készítünk egy `Dolgozo` osztályt, és ebből örököltetjük az állatokat, később az embereket jelképező osztályt is. Ez már egy fokkal jobb legelső megközelítésünknél, de még mindig nem helyes. Egy alkalmazás osztályhierarchiája a valóság egy képe. Ha létrehoznánk egy ilyen nevű közös ősosztályt, az azt jelenthetné, hogy dolgozó az egész világ, és dolgozik benne minden állat és ember. Ne felejtsük el, egyelőre a lovak és a kutyák nem jutalmazhatóak!
Érezzük, hogy az új metódus, ami keresztbe tett a nyarunknak, valamilyen formában önálló osztályként kell, hogy fellépjen. Kényelmes lenne, ha készíthetnénk egy `Dolgozo` osztályt, de ez nem közös ősosztály volna, hanem egy független szerkezet. Ebből a szerkezetből lenne jó örököltetni azokat és csak azokat az osztályokat, amelyeknek jutalom szabható. Ez jelen esetben a `Tehen` osztályra lenne igaz, aminek viszont már van egy őse.
Ha a *C++* nyelv mellett döntöttünk volna a sorozat elején, ez nem lenne gond. Ott ugyanis megengedett a többszörös öröklődés. *Java*-ban ez már nem igaz, itt egy osztály közvetlenül csak egy ősből származhat. Ennek az az oka, hogy problémás annak a helyzetnek a kezelése, amikor egy tag két osztályban is szerepel, és ebből a kettőből örököltetünk. Azért, hogy ez még meggondolatlan tervezés eredményeként se fordulhasson elő, a *Java* nem támogatja a többszörös öröklődést. Amit e szándékosan kifelejtett módszer helyett használhatunk az az interfész.

Interfészek, madárfiókák nélkül

Az interfész hasonló a már megismert osztályokhoz, viszont nélkülözi a változtatható tulajdonságokat, azaz a nem állandó tagváltozókat, és a metódusok törzseit. Ezért tényleg nem más, mint az angol kifejezés, az *interface* magyar jelentése: *felület*. Egy interfész önmagában nem használható, ha nem áll mögötte olyan osztály, amely megvalósítja. A megvalósítás azt jelenti, hogy az interfész által előírt összes metódus törzzsel együtt előfordul az adott osztályban. Lássuk, hogyan fest ez egy *UML* osztálydiagramon (*1. ábra*).
Az ábrán is látható, hogy az interfész a gyakorlatban eléggé közel áll az osztályhoz. Egy hasonló felépítésű doboz jelképezi mindkettőt. Az interfésznek a fejléce azonban dőlt betűs, ami annak elvontabb, idegen kifejezéssel élve absztrakt voltát mutatja. Tagváltozói csak állandók lehetnek, jelen esetben ez a rekesz üres. Műveletei helyén most egyetlen metódus áll, ez a `jutalmaz()`.



1. ábra

Az osztály és interfész közötti fogalmat akkor lehet jól megfogni, ha tisztán modellezési eszközként tekintünk rájuk. Egy osztály tulajdonságok és ezeken végezhető műveletek összessége. Az interfész ezzel szemben annak a leírása, hogy egy osztály milyen műveletekkel kell, hogy rendelkezzen, ha ennek egy megvalósítása. Azt is fontos észrevenni, hogy a megvalósítás jele az *UML* diagramon a szaggatott nyíl, ami megkülönbözteti az örökléstől. Itt ugyanis nem meglévő tagok átvételéről, hanem adott fejlécű műveletek kialakításáról van szó.
Ennek egy fontos következménye, hogy ha készítenénk `Ember` osztályt is, melynek példányait szeretnénk jutalmazni, akkor ebben a `jutalmaz()` metódust külön kell megvalósítani. Ez jól illeszkedik jelen modellünkbe, hiszen valószínűleg másképp szeretnénk jutalmazni egy embert, mint egy tehenet. Azokban az esetekben, ahol ez nem így van, át kell gondolni, hogyan lehetne megoldani a problémát tiszta öröklődéssel.
Lássuk a `Dolgozo` interfész forráskódját. A már megszokott módon, `Dolgozo.java` néven kell elmenteni az alábbiakat:

```
/**
* Ez az interfesz egy olyan dolgozot
* ir le, amelyet meg lehet jutalmazni
* A hogyan kerdese a megvalosito
* osztaly feladata
*/
public interface Dolgozo {
        /**
        * Megjutalmazza a dolgozot.
        */
        public void jutalmaz();
}
```

A kód a megjegyzésekkel együtt is tömör és lényegre törő. Az interfész az azonos nevű kulcsszóval hozható létre. A metódusoknak csupán a fejlécét tartalmazhatja, melyet ; jellel kell lezárni. Nézzük most meg az új `Tehen` osztályt, amely már megvalósítja ezt az interfészt.

```
/**
* Ez az osztaly egy tehenet ir le
* Tulajdonsaga a tej mennyisege, amit egy nap ad.
* Ezt ki lehet iratni a kepernyore.
*/
public class Tehen extends Allat implements
➥Dolgozo {

        /**
        * A naponta adott tej mennyisege.
        */
        private int tej;

        /**
        * Letrehoz egy uj tehenet a megadott
        * hanggal, és napi tej mennyiseggel
        */
        public Tehen(String h, int t) {
                super(h);
                tej = t;
        }

        /**
        * Kiirja a kepernyore a napi
        * tej mennyiseget
        */
        public void mennyiTej() {
                System.out.println(tej + " liter, "
                ➥+ hang);
        }

        /**
        * A tehen megjutalmazasa
        * eseten megszolal es elmosolyodik
        */
        public void jutalmaz() {
                System.out.println(hang + " :-)");
        }

}
```

A változás mindössze az osztály fejlécének kiegészítését és az új metódus bevezetését jelenti. A fejlécben az `implements` kulcsszó mutatja a megvalósítást. Ennek megadásakor, ha elfelejtünk elkészíteni egy tagfüggvényt, az már fordításkor hibát okoz. Az új metódus a megjegyzés és az eddigiek alapján magától értetődő.
Az elmondottak egy kicsit talán rávilágítanak annak a ténynek a miértjére, hogy egy jó programozó egy nap nem ír többet 20 sor kódnál. Rengeteg időt és fáradtságot takaríthatunk meg, ha a vad billentyűtépés helyett először belegondolunk abba, hogy mit szeretnénk elérni. Ezután érdemes elgondolkozni azon, hogy a fejünkben alakuló megoldás eléggé általános-e ahhoz, hogy joggal nevezhessük objektumközpontúnak.
Miután felébredtünk a mély önelemzésből, térjünk vissza teheneinkhez. Mint azt már említettem, az egy osztályból származó egyedek közös vonását kihasználva egyszerű ciklusszerkezettel lépdelhetünk végig az objektumokon. Miért ne tehetnénk ezt meg az ugyanazt az interfészt megvalósító osztályok példányai esetében? Ennél mi sem egyszerűbb, amire jó példa az alábbi, újraírt `main()` függvény:

```
public static void main(String[] args) {
    Tehen tehen1 = new Tehen("múúú", 20);
    Tehen tehen2 = new Tehen("múúú", 30);
    Allat lo = new Allat("nyihaha");
    Allat kutya = new Allat("vauvau");

    Allat[] allatok = new Allat[4];
    allatok[0] = tehen1;
    allatok[1] = tehen2;
    allatok[2] = lo;
    allatok[3] = kutya;

    for (int i = 0; i < allatok.length;
    ➥i ++) {
            allatok[i].szolaljMeg();
    }

    Dolgozo[] dolgozok = new Dolgozo[2];
    dolgozok[0] = tehen1;
    dolgozok[1] = tehen2;

    for (int i = 0; i < dolgozok.length;
    ➥i ++) {
            dolgozok[i].jutalmaz();
    }
}
```

Az alkalmazás fordítása és futtatása után a várt kimenetet láthatjuk:

```
múúú
múúú
nyihaha
vauvau
múúú :-)
múúú :-)
```

Most csak két tehenünket illettük jutalommal, de a dicséretet osztó ciklus ennél sokkal többre képes. Ha a `Dolgozo` interfészt megvalósító osztályok tömbje, a `dolgozok` 1000 elemből állna, és akár az összes objektum más-más osztály példánya volna, akkor is működne az alkalmazás ennek a programrészletnek a módosítása nélkül. Az interfész ugyanis biztosítja azt az egységes felületet, hogy mindnek van `jutalmaz()` metódusa.

Egy hétköznapi példa

Mivel elképzelhető, hogy az Olvasók egy része nem teljes munkaidőben foglalkozik lovakkal, kutyákkal, és tehenekkel, egy gondolatkísérlet erejéig próbáljunk meg egy, az üzleti életre végletekig kihegyezett világunkhoz közelebb álló problémát modellezni. Tegyük fel, hogy egy olyan alkalmazás készítése a célunk, amelyben egy cég munkavállalóit kell kezelni. Erre a jelen cikksorozaton edzett programozó azonnal rávágja, hogy természetesen minden alkalmazott önálló objektum kell, hogy legyen. Ez egy jó megközelítés, hiszen minden dolgozónak van neve, életkora, munkaköre, fizetése, amik tekinthetők az objektum tulajdonságainak. Emellett minden dolgozónak lehet feladatot kiadni, szabadságra küldeni, illetve cégünk humán erőforrás politikájától függően bért emelni, illetve csökkenteni. Ezek pedig az objektumon végezhető műveletek.

Eldöntöttük tehát, hogy mi fog objektumnak számítani. Noha a válasz itt kézenfekvő volt, a tervezésnek ez a lépése nem mindig ilyen könnyű, ezért mindig éljünk a legnagyobb alkotói szabadsággal, és gondoljuk meg, mit nyerhetnénk, ha másképp fogalmaznánk meg a modellt. Adott esetben elképzelhető, hogy az egyes feladatköröket érdemes objektumoknak tekinteni. Most azonban maradjunk az egy dolgozó - egy objektum meggondolásnál, ami a valóságnak is egy jó képe. Miután az objektumok osztályok példányai, meg kell fontolni, hogy milyen osztályhierarchiát szeretnénk kialakítani. Elérkeztünk az első buktatóhoz. Elsőre eszünkbe juthat, hogy van a vállalatnak egy belső struktúrája, ahogy a dolgozók egymás fölé vannak rendelve, használjuk ezt osztályhierarchiának. Az objektumközpontúság tényleg egyfajta leírása a világnak, amiben élünk, de nem szabad az életben használt modelleket gondolkodás nélkül ráerőltetni alkalmazásainkra.

Több, mint valószínű, hogy alkalmazásunkban egy főnök objektumon többféle műveletet végezhetünk, mint egy egyszerű dolgozón. Visszatérve a humán erőforrás politikához, lehet, hogy fizetést csak a főnök objektumok esetén lehet emelni. Ebben az esetben hibás lenne a vállalat hierarchiáját átvenni, hiszen ha a dolgozó kiterjeszti a főnököt, akkor neki is önműködően meglesz az összes metódus az ősosztályból.

A legelső megoldási kísérlet kudarca ellenére érezhetjük, hogy nem járunk messze az igazságtól. Van valami köze a kettőnek egymáshoz, csak nem ugyanazok. Gondoljuk meg, hogy egy vállalat szervezeti felépítésének diagrammján a felelősség lentről felfelé nő, a lehetőségek is ebben az irányban szélesednek. Egy ***UML*** diagramon az ősosztály jellemzően az öröklő osztály felett áll, ami pont fordított irányt mutat az előzőhöz képest. Egy, a ***Balaton*** partjáról érkező szellő azt súgja, hogy fordítsuk meg a szervezeti felépítés diagrammját, és így fejjel lefelé nézve pont megadja a megoldást.

Így azonban többszörös öröklődésbe futunk, ami, mint tudjuk nem megengedett Java-ban. A megközelítés tehát jó, csak meg kell fontolnunk, mit tehetünk meg interfésznek, és mit valódi osztálynak. A sorozatban most először egy próba erejéig megteszem ezt házi feladatnak. A probléma nem túl konkrét, így ki-ki saját ízléséhez mérten bonyolíthatja. E-mailben várom tehát a különféle dolgozók fizetésének, korának, szabadsága időpontjának kiíratását.

Ha a kedves olvasó elakadt valahol, akkor is írjon bátran, szívesen segítek.

Ha már ennyire szép formájú megjegyzéseket készítettünk eddig, következő hónapban kipróbáljuk a ***javadoc***-ot, és próbára tesszük a ***Java*** kivételkezelését is.

Fülöp Balázs (bigwig42@gmail.com)
21 éves, imádja a Túró Rudit, a Debian Linuxot és a teheneket. Kedvenc írója Slawomir Mrožek. Leginkább a számítógépes hálózatok biztonsága érdekli. A BME VIK műszaki informatikus szak hallgatója.

Hálózatok (20. rész)
ARP, RARP, BOOTP, DHCP, CDIR

Folytatjuk tovább az internet vezérlő protokolljainak ismertetését, majd rátérünk az IP protokoll egyre égetőbb problémájára: a még szabadon kioszthatóIP címek vészes fogyására.

Sorozatunk előző részében megismerkedtünk az *IP* címekkel, amelyek lehetővé teszik a gépek egyértelmű azonosítását az interneten. Felmerül azonban egy probléma: a gépek hálózati csatolói mit sem tudnak ezekről a címekről, ők ugyanis egy alacsonyabb szinten, az adatkapcsolati rétegben működnek.
Emlékezzünk vissza, hogy az adatkapcsolati réteg felelős azért, hogy egy *LAN*-on belül az állomások üzeneteket küldhessenek egymásnak keretek formájában. Mivel a kommunikáció nyílt csatornán zajlik, ezért az összes gép megkapja az összes keretet, majd ebből kell kiválogatniuk azokat, amelyek nekik szólnak. Így minden keret esetében meg kellett mondanunk a célpont hálózati csatolójának *MAC* címét. Ez például az *Ethernet* esetében egy 48 bites azonosító, amely a világ összes *Ethernet* hálózati kártyájában különböző.
Amikor az útválasztó az alhálózattól megkap egy olyan csomagot, amely egy, a saját hálózatán belüli géphez tart, akkor a kérdéses csomagot ki kell „bontania”, majd annak tartalmát az adatkapcsolati réteg segítségével keretek formájában eljuttatnia a címzetthez. Az adatkapcsolati réteg azonban egy *MAC* címet vár, a beérkezett csomag fejléce azonban *IP* címet tartalmaz. Az útválasztónak tehát ki kell derítenie az adott *IP* címhez tartozó hálózati csatoló *MAC* címét. Ehhez a legegyszerűbb módszer az, ha az útválasztó „bekiabál” a *LAN* csatornájába, hogy az adott *IP* cím gazdája jelentkezzen. Erre szolgál az ARP nevű protokoll.

ARP (Address Resolution Protocol)

Az *ARP* protokoll tehát az *IP* címek *MAC* címekre történő leképezésére szolgál. Működése nem túl bonyolult. Ha egy gép érdeklődni szeretne például a 213.178.107.36 *IP* címhez tartozó *MAC* cím iránt, akkor csak küld egy adatszórásos üzenetet, amelyben megkérdezi: „Kié a 213.178.107.36 IP cím?”. Erre az üzenetre minden gép reagálni fog, és összehasonlítja saját IP címét a kérdésben szereplővel. Amelyik gép *IP* címe egyezik, az visszaküld egy válaszüzenetet, amelyből a kérdező megtudja a keresett gép *MAC* címét.
Ahhoz azonban, hogy az *ARP* igazán hatékony legyen, szükség van némi optimalizálásra. Beláthatjuk, hogy nem igazán járható az az út, ha minden egyes beérkező csomag esetén újra és újra érdeklődünk a *MAC* cím iránt. Mivel a gépek *IP* címei és hálózati csatolói nem változnak túl nagy gyakorisággal, ezért a birtokunkba került *MAC* címeket egy darabig eltárolhatjuk a memóriában. Persze ha mégis bekövetkezik valami változás, akkor arra gyorsan kell reagálnunk, így a gyorsítótárban tárolt információk érvényességének pár percen belül le kell járniuk.
A hatékonyságot tovább növelhetjük azzal, ha egy *MAC* cím után érdeklődő gép az *ARP* keretben a saját *IP* címét is elhelyezi. Ezt a címet pedig a *LAN* összes gépe eltárolhatja a gyorsítótárában, így az *ARP* keretet küldő gép *IP* címe már mindenki számára ismerté válik.
Amikor egy gép elindul, mindig küld egy *ARP* keretet, amelyben a saját *IP* címéhez keresi az ahhoz tartozó *MAC* címet. Ezzel egyrészt a *LAN* gépei tudomást szereznek az újonnan elindult gép *IP* címéről, másrészt így elkerülhető az *IP* cím ütközés. Ha ugyanis válasz érkezne a gép által küldött *ARP* keretre, akkor az azt jelentené, hogy a hálózaton valaki más is rendelkezik ugyanazzal az *IP* címmel. Ilyen esetben a gépnek nem szabad elindulnia.
Mi a helyzet azonban akkor, ha a keresett gép egy másik *LAN*-on található? Tegyük fel, hogy van egy *B* osztályú címtartományunk (például a 172.16.x.x), és a gépeinket két különálló *LAN*-ba szervezzük. A két hálózatot egy útválasztó köti össze, amelynek két *IP* címe van, tehát mindkét *LAN*-nak tagja.
Tegyük fel, hogy az A gép üzenetet szeretne küldeni a *C* gép számára, ám nem ismeri annak *MAC* címét, ezért egy *ARP* keretet bocsát a hálózatra. Ezt a keretet azonban a *C* gép sosem fogja megkapni, mivel az útválasztó nem továbbítja az adatkapcsolati réteg adatszórásos kereteit.
Erre a problémára két megoldás is létezik. Az első az, hogy az útválasztó válaszol azokra az *ARP* kérésekre, amelyek egy másik hálózatbeli gép iránt érdeklődnek. Ha tehát az A gép érdeklődik a *C* gép *MAC* címe iránt, akkor ennek hatására az útválasztó elküldi *A*-nak a saját *MAC* címét, így az A minden *C*-nek szóló keretét az útválasztóhoz továbbítja. Az útválasztó ezután gondoskodik arról, hogy ezek a keretek eljuthassanak *C*-hez. Ezt a technikát nevezzük ***helyettesítő ARP-nak (proxy ARP)***.

Ez a megoldás feltételezi azt, hogy az útválasztó pontosan tudja, mely alhálózatokat szolgálja ki. A másik megoldás szerint ezt a „tudást” inkább a gépekre kéne bízni, azaz fel kell ismerniük, hogy a keresett gép egy másik hálózatban van. Ebben az esetben az összes forgalmat egyből az útválasztó felé irányítja.

RARP, BOOTP, DHCP

Láttuk, hogyan tudjuk az *IP* címeket leképezni *MAC* címekre. Időnként azonban ezt fordítva is meg szeretnénk tenni: egy ismert *MAC* címhez keressük a hozzá tartozó *IP* címet.
Erre a legjobb példa az, amikor egy olyan munkaállomást indítunk, amely nem rendelkezik háttértárolóval, így magát az operációs rendszert is a hálózatról tölti be. Amikor a gép elindul, akkor fel kell vennie a kapcsolatot a fájlszerverrel, ahonnan letölti az operációs rendszert tartalmazó képfájlt. Ehhez azonban a gépnek meg kell tudnia az *IP* címét.
Erre szolgál a ***RARP (Reverse Address Resolution Protocol)***, amely tulajdonképpen az *ARP* fordítottja, csak itt a *MAC* címet adja meg, és az ahhoz tartozó *IP* címre kíváncsi. Amikor a *RARP* szervert futtató gép meglátja ezt a kérést, akkor egy táblázatból kikeresi, mi az adott géphez rendelt *IP* cím, majd azt visszaküldi.
A *RARP*-nál is felmerül az a probléma, hogy az útválasztók nem továbbítják az adatkapcsolati réteg szintű adatszórásos forgalmat, tehát minden *LAN*-nak rendelkeznie kéne saját *RARP* szerverrel. E probléma megoldására jött létre a ***BOOTP (Bootstrap Protocol)***, amely adatkapcsolati réteg szintű keretek helyett *UDP* csomagokat használ.
A *BOOTP* működése nagyon egyszerű. Amikor a gép elindul, a 255.255.255.255-es *IP* címre (amely az adatszórásra fenntartott speciális *IP* cím) egy *UDP* csomagot, úgynevezett *BOOTREQUEST* csomagot küld, amely tartalmazza a gép *MAC* címét. Ezután vár a válaszra, ha az nem érkezik meg egy bizonyos időn belül, megismétli a *BOOTREQUEST* csomagot.
A *BOOTP* kiszolgáló a *BOOTREQUEST*-re válaszul egy úgynevezett *BOOTREPLY* csomagot küld, amely tartalmazza a gép *IP* címét, a képfájlt tároló fájlszerver címét, valamint az alapértelmezett átjárót és az alhálózati maszkot. Ezután a gép kapcsolatba lép a fájlszerverrel, és például a ***TFTP (Trivial File Transfer Protocol)*** protokoll segítségével letölti az operációs rendszert tartalmazó képfájlt.
Manapság azonban már szinte az összes számítógép rendelkezik háttértárral, így önállóan is képes betölteni az operációs rendszerét, csak a hálózati beállításokat kell „letöltenie”. Ezért a *BOOTP*-t továbbfejlesztették úgy, hogy az alkalmasabb legyen az olyan számítógépek konfigurálására, amelyeket gyakran hordoznak hálózatok között (például laptopok), ezzel megkönnyítve a felhasználók és a rendszergazdák életét. Erre szolgál a ***DHCP (Dynamic Host Configuration Protocol)***.
A *DHCP* ugyan felülről kompatibilis a *BOOTP*-vel, mégis számos dologban különbözik tőle. Ezek közül a legfontosabb az, hogy a *BOOTP* esetében a gépekhez történő *IP* címhozzárendelés általában statikusan történik, azaz egy adatbázisban előre meghatározzuk, hogy melyik gép melyik *IP* címet kaphatja meg. A *DHCP* esetében a címkiosztás általában dinamikus, tehát legtöbbször csak egy halmazt definiálunk, amelyből kiosztjuk a címeket a gépek számára. Egy állomás bekötése egy *DHCP*-t használó hálózatba igazából nem több, mint a hálózati kábelt a géphez csatlakoztatni.

Többesküldés (multicasting)

Bizonyos alkalmazásoknak szüksége lehet arra, hogy egy csomagot egyszerre több címzetthez is eljuttathasson. Ilyen például egy videokonferenciát megvalósító program. Az adatszórás, mint megoldás, természetesen szóba sem jöhet, hiszen a csomagok olyan gépekhez is eljutnának, akik nem is érintettek a kommunikációban (nem beszélve arról, hogy az útválasztók általában nem továbbítják az adatszórásos csomagokat). Az ***unicast***, azaz amikor a csomagot egyesével elküldjük az összes címzetthez sem bizonyul mindig hatékonynak. Tegyük fel, hogy a videokonferencia két résztvevője ugyanabban a távoli hálózatban van. Az ***unicast*** használatával két ugyanolyan csomagot küldünk a hálózat felé, ami erőforrás-pazarló. Sokkal hatékonyabb lenne, ha csak egy csomagot küldenénk, amit a kommunikációban résztvevő mindkét fél megkapna.
Erre jelent megoldást a ***többesküldés (multicasting)***, amely lehetővé teszi, hogy egy csomagnak több címzettje is lehessen. A többesküldésről technikájáról már ejtettünk pár szót a hálózati réteg elméleti tárgyalásakor. Most azt tekintjük át, miként is működik ez az Internet esetében.
A többesküldés megvalósításának első kérdése az, hogy miként lehet összehozni azt az *IP* protokollal. Erre szolgálnak a speciális *D* osztályú *IP* címek ***(224.0.0.0 – 293.255.255.255)***. Minden, ebbe a csoportba tartozó *IP* cím egy többesküldési-csoportot határoz meg, így a kommunikációban résztvevő gépek külön *IP* címmel rendelkezhetnek. Könnyen kiszámolható, hogy egyszerre 250 millió csoport létezhet, amely az internet mai méretét tekintve (lásd később) nem tűnik túl soknak, minden estre ma még elégnek bizonyul.
A többesküldési-csoportoknak két fajtája van: az állandó és az ideiglenes. Az állandó csoportok olyanok, amelyek folyamatosan léteznek. Ilyen például a 224.0.0.1, amely a *LAN*-on belüli összes gépet tartalmazza. Szintén állandó csoport a 224.0.0.2, amelynek a *LAN*-on lévő útválasztók a tagjai.
Az ideiglenes csoportok viszont nem léteznek állandóan, ezeket használat előtt mindig létre kell hozni. A csoportokhoz a gépek szabadon csatlakozhatnak, illetve kiléphetnek onnan. Egy csoport akkor szűnik meg, amikor már egy gép sem csatlakozik hozzá.
A többesküldést speciális útválasztók valósítják meg, amelyek gyakran egybe vannak építve a „közönséges” útválasztókkal. Ezek bizonyos időközönként megkérdezik a hozzájuk kapcsolódó hálózat gépeit (általában többesküldéssel, a 224.0.0.1-es címen keresztül), hogy mely többesküldési csoportoknak a tagjai. Ezt az úgynevezett ***IGMP (Internet Group Management Protocol – internet csoportfelügyeleti protokoll)*** segítségével végzik, amely az *ICMP*-hez nagyon hasonló vezérlőprotokoll. A többesküldéses útválasztók feszítőfák felállításával jelölik ki a többesküldéses csomagok útvonalát (lásd sorozatunk 17. részét).

CIDR (Classless InterDomain Routing)

Az *IP* jelenlegi verziója (az *IPv4*) jól bevált, kiforrott protokoll, bizonyítékul szolgál erre az Internet hihetetlen méretű terjedése. Ironikus módon a hálózat nagyléptékű

növekedése fogja az *IPv4* vesztét is okozni: a szabad (még ki nem osztott) *IP* címek száma vészesen csökken. Pontosabban nem is a szabad *IP* címek fogynak, hiszen abból több mint 4 milliárd létezik. A gond a címek tartományokra való tagolásából adódik. Mint utóbb kiderült, elég szerencsétlen módon sikerült az osztályokat megválasztani: az A osztályú tartományok a legtöbb hálózat számára túl nagy, a *C* osztályúak pedig túl kicsik. Sok intézmény tehát érthető módon *B* osztályú tartományokért folyamodott. Egy felmérésből azonban kiderült, hogy a *B* osztályt birtokló hálózatok jelentős része nem is használja ki igazán címtartományát, akár egy *C* osztályú címcsoporttal is vígan működhetne. Az üzemeltetők mégis egy *B* csoportot kérvényeztek, hiszen így nem okoz majd kellemetlenséget, ha a gépek száma egyszer majd 254 fölé emelkedik. Így azonban *IP* címek milliói mennek veszendőbe.

Ez a probléma valószínűleg elkerülhető lett volna azzal, ha a *C* osztályú címeknél 8 helyett 10 bitet szánunk a gépek azonosítására. Ez már elegendő lett volna a hálózatok többségének, így tényleg csak az folyamodott volna *B* osztályú tartományért, akinek tényleg szüksége is lett volna rá.

A most használt címtartományok másik problémája az útválasztók táblázatainak növekedésében jelentkezik. Az útválasztók számára az *IP* címek két részből állnak: a számukra teljesen érdektelen gépcímből, és a hálózatcímből. Az útválasztóknak minden hálózatcímhez fenn kell tartaniuk egy bejegyzést a forgalomirányító táblázatukban, amely megadja, hogy melyik kimeneten kell az arrafelé tartó csomagot továbbítani.

Ahogy egyre gyarapszik az Internethez kapcsolt hálózatok száma, úgy nőnek az útválasztók táblázatai. A nagyméretű táblázatokkal azonban máshogy kell bánni. Egyrészt több memória is kell hozzá, másrészt komplexebb kereső algoritmusok, amelyek segítségével az útválasztók gyorsabban megtalálhatják a kívánt bejegyzést. Sok, régebbi útválasztó által használt eljárás elavul, így azok az útválasztók nem tudják hatékonyan ellátni feladatukat.

A nagy méretű táblázatokkal azonban más baj is van. Az útválasztóknak folyamatosan cserélgetniük kell egymás között a táblázatuk tartalmait. Kis méretű táblázatok esetében ez könnyen ment, nagy méret esetében azonban a folyamat egyrészt lassú, másrészt sérülékeny. Egy nagyobb adatmennyiség nagyobb valószínűséggel sérül meg egy átvitel alatt. A sérült táblázatok pedig sok galibát okozhatnak a forgalomirányításban.

A forgalomirányító táblázatok azért nőnek ilyen mértékben, mert az *IP* címek nem tartalmaznak semmiféle információt arra nézve, hogy az adott hálózat merre található (nem tükrözik az Internet topológiáját). Ha például kiköthetnénk azt, hogy az *IP* címek első három bitje a földrészt, a következő öt pedig az országot jelölje, ahol az adott hálózat található, akkor a forgalomirányító táblázatban jelentős számú bejegyzést megspórolhatnánk. Sajnos erre a rendelkezésre álló 32 bit már kevés, nem is beszélve arról, hogy ez a megoldás nem használná ki hatékonyan a címeket, hiszen a kisebb országok is ugyanannyi címet kapnának, mint a nagyobbak.

Sajnos erre a két problémára nem létezik igazi megoldás (illetve létezik: az egész *IPv4* lecserélése egy újabb verzióra). Csupán annyit lehet tenni, hogy bizonyos intézkedésekkel időt nyerünk, de előbb vagy utóbb ismét szembekerülünk ezekkel a gondokkal.

Az egyik ilyen intézkedés, amely segítségével elódázható az Internet problémáinak megoldását a ***CIDR (Classless InterDomain Routing – osztály nélküli tartományok közötti forgalomirányítás)***, amely „megreformálja” a *C* osztályú *IP* címtartományok kiosztását. Először is, akinek nincs szüksége egy teljes *B* osztályú tartományra, mert például csak 400 gépet szeretne az Internethez kapcsolni, akkor az kap egy 512 darab címből álló egybefüggő címblokkot (két összefüggő *C* osztályú címtartományt). Akinek több gépe van, például 2000, az nyolc darab *C* osztályú tartományt kap.

A kiosztásnál azonban azt is figyelembe veszik, hogy az adott hálózat a világ melyik pontján található. Ha például ***Európában***, akkor a címeket a 194.0.0.0 – 195.255.255.255-ig terjedő tartományból osztják ki. Ezzel csökkenthetjük a forgalomirányító táblázatok méreteit, hiszen ha az útválasztó egy ilyen címmel találkozik, akkor egyből az alapértelmezett európai átjáróhoz küldheti.

Persze az európai útválasztóknak részletes ismeretekkel kell rendelkezniük az európai hálózatokról. Felmerül a kérdés, hogy mivel felrúgtuk az eddig használt *IP* címosztályokat, az útválasztók miként döntik el egy csomagról, hogy azt mely hálózat útválasztójához kell továbbítania? Továbbá nem áll-e fent megint a forgalomirányító táblázatok méretének megugrásának veszélye?

Az utóbbi kérdésre a válasz az, hogy nem tartunk fent minden *C* osztályú *IP* címnek külön bejegyzést, hiszen a *CIDR* használatával örökre búcsút mondunk az *IP* címek ily módon történő felosztásának. Most már mi határozzuk meg, hogy az *IP* címből hány bit szolgáljon a hálózat azonosítására. A forgalomirányító táblázatban csak a kiosztott címcsoport alapcímét és annak 32 bites maszkját tároljuk el. (Például egy intézmény igényt tart 2048 darab *IP* címre, így megkapja a 192.24.0.0 – 194.24.7.255-ig tartó címeket. A maszk azt mondja meg, hogy hány bit szolgál a hálózat azonosítására. Jelen esetben 21 darab bit, így a maszk 21 darab egyesből, és 11 darab 0-ból fog állni. Egy hálózat alapcímén a kiosztott címcsoport első címét értjük, jelen esetben ez a 192.24.0.0).

Tegyük fel, hogy beérkezik egy olyan csomag, amely a 192.24.2.5-ös *IP* címre tart. Ilyenkor az útválasztónak végig kell néznie a forgalomirányító táblája összes bejegyzését, és mindegyiken el kell végeznie a maszkolást, azaz egy logikai ÉS műveletet a csomag célpont címe és az adott bejegyzés maszkja között. Ha az így kapott érték megegyezik az adott hálózat alapcímével, akkor megtaláltuk azt a hálózatot, ahova a csomag tart, így továbbíthatjuk a megfelelő útválasztó felé.

A következő részben megismerkedünk az *IP* protokoll új generációjával, az *IPv6*-al, amely 32 bit helyett 16 bájton tárolja a címeket, kiküszöböli az *IPv4* hibáinak nagy részét, és mindezek mellett sokkal hatékonyabb és rugalmasabb is nála.

Garzó András
garzo@interware.hu

Elfeledett terminálok

A parancssor megszállottai grafikus játékszereket kapnak, melyekkel helyet takaríthatnak meg, háttereket használhatnak vagy akár több terminált is elindíthatnak egyetlen paranccsal.

Kérlek, ***François***, ne sírj. Mi bánt ennyire? Az étterem fantasztikusan néz ki, ***mon ami***. Minden készen áll. Ugyan mi lehet ilyen rossz? ***Quoi?*** Átfutottad a menüt, és láttad, hogy terminálprogramok szerepelnek rajta, ez hangolt le ennyire? Ó, ***Mon Dieu***, ***François***, a terminálok nem valakinek vagy valaminek a végét jelentik, hanem a héjhoz tartozó ablakokat, mint a ***GNOME*** terminál, az ***xterm*** vagy a ***KDE Konsole***. Érkezőben a vendégeink, ***François***. Üdvözlök mindenkit a ***Chez Marcel***-ben, ahol a finom linuxos fogások mellé kiváló borok társulnak. Foglaljatok helyet, helyezzétek magatok kényelembe. ***François***, kérlek menj le a pincébe, és hozd fel az 1999-es ***Côte du Rhône***-t. ***Vite!***

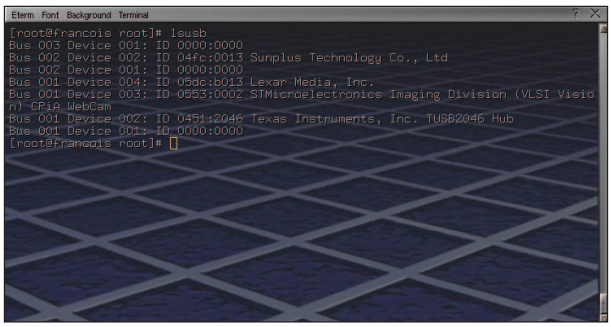
Mai menünk a rendszerfelügyeletről szól, és bizony nem kis gondot okozott az összeállítása, hiszen ezt a témát számtalan szemszögből lehet megközelíteni. Aztán feltettem magamnak a kérdést: melyik az az eszköz, amit a rendszergazdák a legtöbbet használnak? A válasz kézenfekvő volt, a parancshéjat, amely valamilyen terminálemulátorban fut. Bár ma már alapvető eszköznek számítanak, a terminálprogramok komoly fejlődésen mentek keresztül a jó öreg ***xterm*** ideje óta.

Akik még túl fiatalok ahhoz, hogy emlékezhessenek rá, azoknak elmondanám, hogy azért hívjuk őket terminálprogramoknak vagy -emulátoroknak, és nem héjablakoknak, mert eredetileg azoknak a termináloknak a szoftveres, asztali gépekre készült megfelelői voltak, amelyeket régen fizikailag csatlakoztattak a nagygépekhez. Na, míg én mesélek, François meg is érkezett a borral.

Az évek során jó néhány terminálprogramot használtam. Eredetileg csak a sima ***xterm*** volt, ami a legtöbb linuxos rendszeren ma is megtalálható. Ha ki akartuk csicsázni, megváltoztattuk a betűtípus vagy a háttér színét, de másra nem nagyon adott lehetőséget. Ha például az ***xtermet*** acélkék háttérrel és piros betűtípussal akarjuk futtatni, a következő parancsot kell kiadnunk:

```
xterm -background "SteelBlue" -foreground "Red"
```

Később felfedeztem egy másik, ***rxvt*** nevű programot, a legtöbb rendszeren ez is megtalálható. Ez már kicsivel több beállítási lehetőséget kínált. Nemcsak jobban



1. ábra Átlátszóság, hátterek, témák és egyebek Eterm alatt

nézett ki, mint az ***xterm***, de ***XPM*** bittérképet is képes volt háttérként használni, ami abban az időben egész jó dolog volt:

```
rxvt -pixmap
/usr/share/themes/BrushedMetal/gtk/bg.xpm -fg
black
```

Később jött az első olyan terminálemulátor, amit igazán szeretni tudtam, és amit ma is örömmel használok. A neve ***Eterm***, és bár az ***xterm*** helyettesítőjének tervezték az ***Enlightenment*** ablakkezelőhöz, ***GNOME***, ***XFCE***, ***KDE*** vagy bármi más alatt is működnie kell. Érdemes szétnézni terjesztésünk ***CD***-lemezén, mert alapesetben valószínűleg nem települ. Az ***Eterm*** legújabb változatát az ***Eterm*** webhelyről is beszerezhetjük (lásd az internetes forrásokat).

Az ***Eterm*** nemcsak jól néz ki, de tud néhány nagyszerű dolgot is. Például sokféle háttérkép kezelésére képes – sőt, eredetileg is jó néhány tartozik hozzá, ezek egy része csempeszerűen tölti ki a hátteret, mások teljes méretűek. Ha ilyet szeretnénk látni a képernyőnkön, csak kattintsunk az ***Eterm*** menüsorának ***Background (Háttér)*** majd ***Pixmap (Bittérkép)*** elemére, majd válasszunk ki egyet a rendelkezésünkre álló képek közül.

Ugyancsak módosíthatjuk a betűtípus méretét, a fényerőt, a képélességet, a gördítősáv stílusát és sok egyebet.

Ha kívánjuk, saját, testreszabott menüt is létrehozhatunk. Ha elégedettek vagyunk a módosított beállításokkal,

kattintsunk a menüsor ***Eterm***, majd ***Save User Settings (Felhasználói beállítások mentése)*** elemére.
Az ***Eterm*** témákkal is felöltöztethető. Kattintsunk az ***Eterm*** webhelyén található ***Themes (Témák)*** hivatkozásra, és számtalan téma tárul elénk. A tetszésünket elnyerőket ***.tar*** és ***.gz*** csomagok formájában szabadon letölthetjük és telepíthetjük. Telepítésükhöz hozzunk létre egy .Eterm könyvtárat a `$HOME` könyvtárunkban, majd ez alatt egy ***themes*** könyvtárat. Az egyes témákat elég kibontanunk ide, máris elérhetőkké válnak. Ha az ***Etermet*** a következő alkalommal már a kiválasztott témával akarjunk indítani, akkor a következő parancsot kell kiadnunk:

```
Eterm -t téma_neve
```

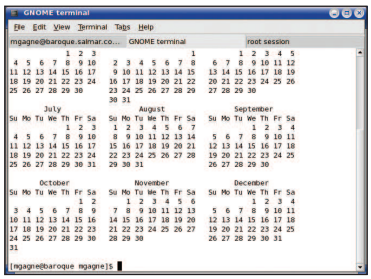
Én a legjobban mégis azt szeretem az ***Etermben***, hogy átlátszóvá lehet tenni. Különösen azért tetszik ez a szolgáltatás, mert így figyelemmel kísérhetem a háttérben gördülő rendszernaplókat, és persze a kedvenc háttérképem is látható marad. Ehhez csak annyit kell tennünk, hogy rákattintunk a menüsor ***Background (Háttér)*** elemére, majd engedélyezzük a ***Toggle Transparency (Átlátszóság átkapcsolása)*** beállítást. A kapcsoló ***KDE*** alatt nem működik, de ***GNOME***, ***WindowMaker*** és egyebek alatt igen.
Ha például az Apache naplóit akarom figyelni a webkiszolgálómon, és átlátszó ***Etermet*** akarok futtatni, a következő parancsot szoktam kiadni:

```
Eterm -O -e sudo tail -f
/var/log/httpd/access_log
```

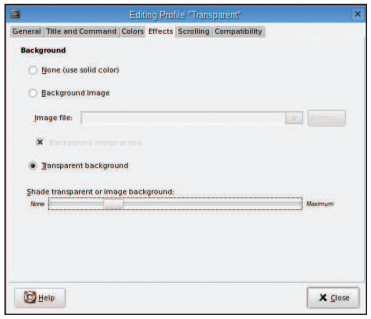
A terminálemulátorokhoz ugyan nem sok köze van, de megjegyezném, hogy a fenti parancs futtatásához hozzá kell adnunk magunkat a ***/etc/sudoers*** fájlhoz – kivéve persze azt az esetet, ha rootként futtatjuk az ***Etermet***. Én az alsó sort írtam hozzá a fájlhoz, közvetlenül a root jogainak meghatározása alá:

```
# User privilege specification
➥(Felhasználói jogok meghatározása)
root    ALL=(ALL) ALL
marcel  ALL=(ALL) ALL
```

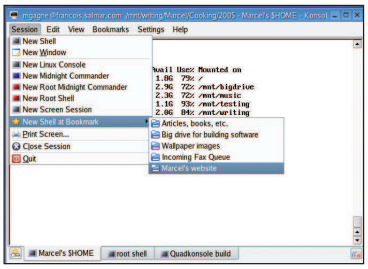
A terminálprogramok újabban egy másik izgalmas szolgáltatást is támogatnak, és ez a lapok kezelése. Lapok alkalmazásával nincs szükség több terminálprogram elindítására, vagyis jóval kevesebb felületre van szükség a képernyőn.



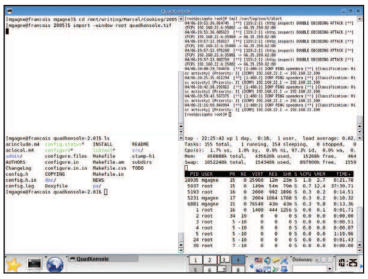
2. ábra A GNOME terminál lapjai révén takarékoskodhatunk a képernyő felületével



3. ábra Ha átlátszó GNOME terminált akarunk, először egy profilt kell létrehoznunk



4. ábra Konsole-lapok, könyvjelzők és egyebek

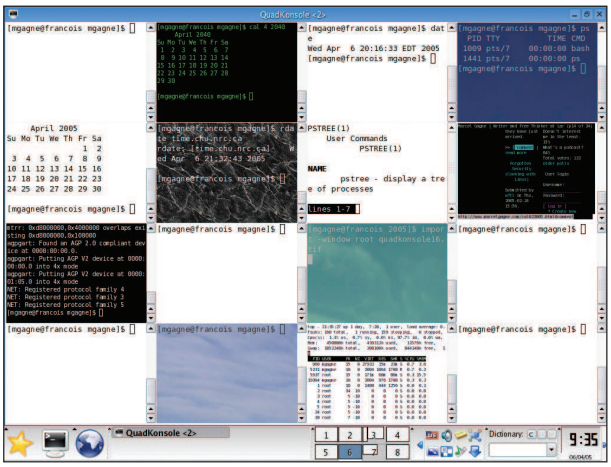


5. ábra A Quadkonsole négy Konsole-t indít egy rácson

Vessünk egy pillantást például a ***KDE Konsole*** programjára, vagy akár a ***GNOME*** terminálra. Mindkettőre igaz, hogy nemcsak alapszintű héjelérést biztosít, de lapokra kerülő munkamenetekben több héj futtatását is lehetővé teszik. Ha a ***GNOME*** terminálban több lapot akarunk használni, nyomjuk meg a SHIFT-CTRL-T billentyűkombinációt, és máris megnyílik egy új munkamenet (*2. ábra*).
A ***GNOME*** terminál ugyancsak támogatja az átlátszóságot, vagyis ha olyan parancsfájlt akarunk indítani, amely a kedvenc háttérképünk felett gördíti a naplók tartalmát, ezt is megtehetjük. A ***GNOME*** terminál esetében ezek a szolgáltatások profilok révén érhetők el. Ha létre szeretnénk hozni egy profilt, kattintsunk a ***GNOME*** terminál menüsorának ***Edit (Szerkesztés)*** elemére, majd válasszuk a ***Profiles (Profilok)*** parancsot. Kattintsunk a ***New (Új)*** elemre, majd adjunk a profilnak egy nevet, mint például „atlatszo”. Miután az ***Editing Profile (Profil szerkesztése)*** párbeszédpanel megjelent, kattintsunk az Effects (Hatások) fülre, majd a ***Transparent background (Átlátszó háttér)*** választógombra (*3. ábra*).
A csúszka segítségével állítsuk be az átlátszóság szintjét, majd zárjuk be a profilablakot. Ez után a következő paranccsal indíthatjuk el a ***GNOME*** terminált:

```
gnome-terminal --window-with
➥-profile=atlatszo \
-e “sudo tail -f
➥/var/log/messages”
```

Ha tetszik az eredmény, a fenti parancsot írjuk be egy parancsfájlba, aminek persze adjunk egy általunk értelmesnek tartott nevet.
A ***KDE*** világában a ***Konsole*** áll rendelkezésünkre, mely szintén kiválóan kezeli a lapokat. Ha új lapon szeretnénk ***Konsole*** munkamenetet indítani, kattintsunk a menü ***Session (Munkamenet)*** elemére, majd válasszuk a ***New Shell (Új héj)*** parancsot. A lapokat duplán rájuk kattintva, majd névvel ellátva szabhatjuk testre.
Ha több héjat is futtatunk – például az egyik helyen programot fordítunk, a másikban figyelünk valamit, a harmadikban pedig rootként tevékenykedünk –, akkor a lapok elnevezése fontos kényelmi szolgáltatás lehet. Megjegyezném még, hogy a ***Session*** menüre kattintva számos egyéb beállítást és lehetőséget is elérhetünk, mint például a könyvjelzővel jelölt helyen történő munkamenet-indítás (*4. ábra*).



6. ábra Ha túl sok információt kellene befogadnunk, a Quadkonsole segítségünkre siet

Könyvjelzők? Miközben rendszergazdai teendőinket végezzük, egy idő után észrevesszük, hogy újra és újra ugyanazokba a könyvtárakba kell belépnünk. Persze, gyorsan gépelünk, így bárhova pillanatok alatt eljutunk, könyvjelzőkkel mégis könnyebb. Amikor éppen a könyvtárfa sokadik szintjén lévő, mégis sokszor használt könyvtárban vagyunk, kattintsunk a menüsorra, majd az ***Add Bookmark (Könyvjelző hozzáadása)*** parancsra. A megjelölt helyhez nevet is rendelhetünk, majd a későbbiekben egyetlen kattintással megnyithatunk egy héjat az adott könyvtárban.
Természetesen nem léphetünk tovább úgy, hogy az átlátszóságról ne essék szó. A ***Konsole***-ban az átlátszóságot a menüsor ***Settings (Beállítások)*** elemére és a ***Schema (Séma)*** parancsra kattintva, majd az egyik átlátszóságot kiválasztva engedélyezhetjük. Miközben a ***Settings*** menüben mozgunk, bizonyára észre fogjuk venni, hogy a ***Konsole*** terminálprogram számtalan beállítással rendelkezik. Megváltoztathatjuk a betűkészletet, átválthatunk teljes képernyős módra, megváltoztathatjuk a kódolást és a billentyűzet kiosztását, továbbá riasztásokat adhatunk meg arra az esetre, ha a lapokon futó munkamenetekben valamilyen változás állna be. Senkit nem fosztanék meg a ***Konsole*** önálló megismerésének örömétől, de hadd mutassak egy parancsfájlt, amivel letisztult, átlátszó ***Konsole***-t indíthatunk a naplófájl figyelésére:

```
konsole --schema Transparent.schema --nomenubar \
  --notabbar --noframe \
  -e sudo tail -f /var/log/messages
```

Mai menünk utolsó fogása egy nagyszerű program lesz – egész egyszerűen nem is értem, korábban hogyhogy nem láttam még hasonlót sem. Miután felfedeztem, azonnal rendszeresen használni kezdtem, és azóta sem akarok megválni tőle. A ***KDE kpartsra*** építve ***Simon Perreault*** ***Quadkonsole*** névvel új programot írt. Mint neve is utal rá, négy ***Konsole*** programot indít el egy rácsozatban (*5. ábra*); így nincs szükség arra, hogy a ***Konsole*** ablakait egymás mellé rendezzük vagy a héjak között a lapokra kattintva váltogassunk. További szépsége, hogy a ***Quadkonsole*** a tálcán egyetlen programként jelenik meg, vagyis a ***Kicker*** panel áttekinthető marad. Ez különösen hasznosnak bizonyul majd, ahogy tovább ismerkedünk a programmal.
Mindegyik futó ***Konsole***-t kívánságunk szerint módosíthatjuk, ehhez rá kell kattintanunk a jobb oldali egérgombbal, ki kell választanunk a ***Settings (Beállítások)*** parancsot, majd végre kell hajtanunk a megfelelő változtatásokat. Megtehetjük például, hogy az egyik ablakot átlátszóra állítjuk, a másikban fekete háttéren kisebb méretű, fehér betűkkel írunk és így tovább. Tulajdonképpen mindegyik ablak egy önálló ***Konsole*** példány, és beállításait is ennek megfelelően változtathatjuk.
Ha ez sem elég, a ***Quadkonsole***-t négynél több ***Konsole***-lal is indíthatjuk, ehhez csak meg kell adnunk a sorok (`rows`) és az oszlopok (`columns`) számát:

```
quadkonsole --rows 4 --columns 4
```

A fenti parancs hatására a ***Quadkonsole*** pontosan 16 különálló terminálkapcsolattal indul (*6. ábra*). Éppen ez az, amikor a ***Kicker*** panel tisztán tartása jól jön. Nyilván, ha azt akarjuk, hogy az információk tényelegesen el is férjenek a képernyőnkön, akkor érdemes eljátszanunk a betűméretekkel. Nálam például az egyik ablakban a top egy különösen kisméretű betűtípussal valós időben figyeli a folyamatokat és az erőforrások használatát. Igaz, ha látni akarok belőle valamit, elő kell vennem a nagyítót – erre van például a ***kmag***.
Alapesetben a fókusz követi az egérmutatót, ebből furcsa balesetek származhatnak, ha gépelés közben véletlenül meglökjük az egeret. Az alapbeállítást érdemes tehát felülbírálni, erre a ***Quadkonsole*** indításakor használható `--clickfocus` kapcsoló használható. Alkalmazásakor egy-egy ***Konsole*** kiválasztásához rá kell kattintanunk a megfelelő ablakra.
Én is a kattintásos fókuszváltást szeretem, csakhogy sokszor idegesítő tud lenni, ha folyton az egérért kell nyúlkálni. Szerencsére a ***Quadkonsole*** erre is kínál megoldást, a ***Konsole*** példányok között a SHIFT-CTRL-kurzorgombok kombinációval tudunk fel, le, jobbra és balra váltogatni.
Incroyable! Úgy tűnik, ***mes amis***, ismét elérkezett a záróra. Bár tudna valaki írni egy olyan programot is, ami szabadidőnk egyetlen órácskájából négyet, vagy akár 16-ot varázsolna! Itt a rengeteg finom bor és a számtalan kiváló program, volna mit ízlelgetnünk, ha időnk engedné. Semmi baj, ***mes amis***, egy kis idő még belefér, és csevegés közben ***François*** örömmel tölti újra poharatok. Dőljetek hátra, és élvezzétek a ***Côte du Rhône*** zamatát! Emeljük fel poharunkat, ***mes amis***, és igyunk egymás egészségére!
A ***votre santé! Bon appétit!***

Linux Journal 2005. július, 135. szám

A cikkhez tartozó források elérhetősége:
➲ ***www.linuxjournal.com/article/8259***

Marcel Gagné (mggagne@salmar.com) Mississaguában, Ontario államban él. Ő a szerzője a Kiskapu kiadásában tavaly szeptemberben megjelent Linux-rendszerfelügyelet (ISBN 96-9301-40) című könyvnek.

Egy szabadalmi irányelvtervezet „tündöklése” és bukása

2005. július 6. emlékezetes nap lesz a szabad szoftverek történelmében. A mai napon ugyanis 648 nem szavazattal, 14 ellenében, 18 tartózkodás mellett az Európai Parlament elvetette a számítógéppel megvalósított találmányok szabadalmazására vonatkozó (ismertebb nevén: szoftver-szabadalmi) irányelvtervezetet.

Egyszer volt, hol nem volt....

A történet 1973-ban kezdődik, az Európai Szabadalmi Egyezmény müncheni aláírásával, mikoris az egyezmény szövegébe bekerül az alábbi részlet:

> *„52. cikk A szabadalmazható találmányok*
> *(2) Nem minősül az (1) bekezdés szerinti találmánynak különösen:*
> *... c, a szellemi tevékenységre, játékra, üzletvitelre vonatkozó terv, szabály vagy eljárás, valamint a számítógépi program....*
> *(3) a (2) bekezdésben felsoroltak szabadalmazhatósága csak annyiban kizárt, amennyiben az európai szabadalmi bejelentés vagy az európai szabadalom* ***rájuk kizárólag e minőségükben*** *vonatkozik.”*

A vastagon szedett szavak ***Németország*** közbenjárására kerültek az irányelv szövegébe, habár senki sem tudta pontosan, mit is kéne érteni a ***„számítógépi program, mint olyan”*** kifejezés alatt. Az évek azonban megmutatták, hogy az ***Európai Szabadalmi Hivatal*** hajlamos arra az értelmezésre, hogy e kizáró tényező alkalmazását a lemezen vagy egyéb hordozón benyújtott kódra alkalmazza, amennyiben azt semmi egyéb dokumentáció vagy leírás nem kísérte. Amennyiben azonban a kacifántos körmondatokban megfogalmazott igénypontokból azt lehetett kivenni, hogy a szoftveres megoldás valójában egy ártatlan eljárási szabadalom, ami ezáltal nem eshet kizárás alá, hiszen számítástechnikai, műszaki eljárások szabadalmazása egyáltalán nem volt tiltott.
Ez vezethetett oda, hogy napjainkban egy német, szabadalmi ügyekkel foglalkozó iroda ügyvédje kijelenthette: ***„bármi, ami tartalmaz újdonságelemet szabadalmazható, csak jól kell megfogalmazni.”*** Figyelemmel arra, hogy voltaképpen valamennyi szoftver ***„eljárás”***, e tulajdonságainak részletezése szakavatottak számára nem jelenthet különösebb problémát.

Lássunk pár példát

A ***Parlament*** javaslatai között szerepelt, hogy az irányelvtervezetet magát nevezzék át, hiszen a számítógéppel megvalósított találmányok félrevezetőek, hiszen azt sejtetik, hogy a találmánynak magán a számítógépen kell megvalósulnia (azaz nyilván a gépen futó szoftverek formájában). Ezért a javasolt fogalom: ***„számítógéppel vezérelt”*** találmányok lett volna.
A ***Tanács*** például nem támogatta azt a megoldást, mely szerint a szabadalmazás feltételének kellene tekinteni valamely, a fizikai világgal való kapcsolat, ami jelezné, hogy a ***„találmány”*** többre képes, mint holmi adatkezelés.

Az irányelv múltja

Az irányelv iránti igény az ezredforduló körül ütötte fel a fejét, és 2002-ben nyerte el első alakját. Az ***Európai Bizottság*** tehát a tagországok szabadalmi jogának harmonizálását szem előtt tartva megalkotta az irányelvtervezet első változatát és 2002. szeptemberében előterjesztette azt. Miután kézhez kapta, a ***Parlament*** 2003 szeptemberben alaposan teletűzdelte módosító indítványokkal (64 darab került elfogadásra) a szöveget és véleményét megküldte a ***Tanácsnak***, ahol azonban semmibe vették az említett javaslatokat igyekeztek kialakítani az úgynevezett ***„közös álláspontot”***, melynek elfogadásához minősített többségre volt szükség. Az új tagállamok csatlakozása és a módosított szavazatszámok okoztak némi fennakadást (emlékezzünk csak a lengyelek ideiglenes blokkolására 2004. decemberében a ***Mezőgazdasági és Halászati Miniszterek Tanácsában***). 2005. februárjából ismert a Parlament hiábavaló kísérlete,

melyben kezdeményezte a szöveg Bizottság általi visszavonását és újragondolását. 2005. március 7-én sikerült aztán megfelelő többséget találni az irányelv elfogadásához, ehhez kellett egy spanyol ellenszavazat, ***Ausztria***, ***Belgium*** és ***Olaszország*** tartózkodása valamint a többi tagállam – köztük hazánk – igen szavazata.
Ezt követte az utolsó forduló.
Az anyagot megküldték a ***Parlamentnek***, akinek 3 hónapja volt a felkészülésre. Ez alatt 256 módosító indítvány érkezett be, ezek egy része átfedésben egymással.
A legátfogóbb (és szövegében legtámogathatóbb) módosítás csokrot ***Michel Rocard*** nyújtotta be. A tanácsi szöveggel kapcsolatos elégedetlenséget jelezte, hogy az irányelv tervezet majd valamennyi pontját érintette módosítási indítvány.
Érdekes megjegyezni, hogy 2, a kis és középvállalkozások érdekeinek védelmét előmozdító módosítás például magyar képviselőktől származott.

Az irányelvjavaslat sebei

Az irányelvjavaslat szövege egyébként olyan komplikáltra sikerült, hogy okos, és a témában járatos emberek akár órák hosszat el tudtak vitatkozni rajta, avatatlan külső szemlélők pedig nem is merészkedtek a közelébe. Fő hibái, hogy jogászok technikai fogalmakkal próbáltak megküzdeni ráadásul a szabadalmi jog hálójába bonyolódva. Bár eredeti célja a jogterületre vonatkozó harmonizáció lett volna, gyakorlatilag csak összezavarta a szálakat. Az állóvíz felbolydult, és hiába hangsúlyozták a szabadalmi hivatalok képviselői, hogy ez az egész a jelenlegi ***„status quo”*** fenntartására irányuló kísérlet, már senki sem hitt nekik.
Beépítette ugyan a ***Tanács*** a ***„számítógépi program, mint olyan”*** nem szabadalmazható frázist, ugyanakkor félreértések gócpontja ként tartalmazta az alábbi szöveget is:

*„**Nem engedélyezhető** az akár önmagában álló, akár hordozón lévő számítógépi programra vonatkozó igénypont, **kivéve, ha a program** annak programozott számítógépre, programozott számítógépes hálózatra vagy más egyéb programozott berendezésre történő feltöltése és végrehajtása esetén ugyanazon **szabadalmi bejelentésben** az (1) bekezdéssel összhangban **igényelt** terméket vagy **eljárást valósít meg**.”*

(A vastagon szedett részek összeolvasásából látható a szöveg másik arca.)

Felszínre kerültek hibásan bejegyzett szabadalmak, megsajdult a kis- és középvállalatok szíve és pénztárcája, mikor megtudták, hogy az európai szabadalmi eljárás átlagos költsége 29.000 euró (több, mint 7 millió forint). Kibukott, hogy az újdonságkutatások alapját a szoftveróriások által rendelkezésre bocsátott adatbázisok illetve a már bejegyzett szabadalmak kézik, ezért Európaszerte kísérleteket tettek a szabad szoftverek dokumentált közzétételére, hogy bizonyítható legyen bármelyik bíróság előtt, „a levédett szoftver nem új” és így védelemre sem érdemes.
Fontos tudni a szabadalmi oltalomról, hogy rendszeres díjfizetés mellett a védelem 20 éven át megilleti a szabadalmast és valamennyi, a szabadalmat használni vágyó köteles neki licencdíjat fizetni, vagy vállalni a szabadalombitorlással kapcsolatos eljárás megpróbáltatásait.

A Parlament visszavág

Tony Robinson szavait szabadon idézve:

„Jobb, ha nincs irányelv, mintha az ami van, rossz.”

A ***Parlament*** tiszta lapot adott a témának. Időt arra, hogy a ***Bizottság*** és a ***Tanács*** átgondolhassa a történteket és ha eljön az ideje, egy jó, mindenki számára hasznos szöveggel próbálkozzon újra. Hiszen voltak most is jó gondolatok, mint az ***Európai Szabadalmi Hivatal*** feletti bírósági kontroll, vagy a számítógéppel vezérelt találmányok szabadalmazására vonatkozó gondolat.

A háború folytatódik

Hadd idézzem egy kedves barátomat, aki euforikus örömöm hallatán csendesen megjegyezte:

„Ma nyertünk egy csatát.
De a háború folytatódik.”

Ünnepeljünk hát, de tudjuk, hogy harcolni kell majd megint, mindaddig, míg egy korrekt szabályozás nem születik. Most jönnek majd a mindennapok apró csetepatéi, egy-egy csendesen bejegyzett szoftverszabadalom elleni küzdelem. Fegyverbe hát!

Dr. Dudás Ágnes (dudas.agnes@abend.hu) ügyvédjelölt, az FSF egyik aktivistája. 2004-ben végzett az ELTE Jogtudományi Karán. Szakdolgozatát a szoftverek szerzői jogi védelméről írta, a 2003-as évet pedig e terület kutatásával a berlini Humboldt Egyetemen töltötte.