Programkönyvtár HOGYAN

David A. Wheeler

Ez a HOGYAN azoknak a programozóknak készült, akik programkönyvtárakat szeretnének használni Linuxon. Összefoglalja a programkönyvtárak készítését és használatát. Egyaránt tartalmazza a statikus (static), megosztott (shared), valamint a dinamikusan betölthető (dynamically loaded) programkönyvtárakkal kapcsolatos ismereteket.

Tartalomjegyzék

1. Bevezetés

Ez a HOGYAN programozóknak készült, és összefoglalja, hogyan készíthetsz és használhatsz programkönyvtárakat Linuxon, a GNU eszközkészlet felhasználásával. A "programkönyvtár" kifejezés egyszerűen egy olyan fájlt jelöl, ami lefordított tárgykódot (és adatot) tartalmaz, amit később egy programmal össze lehet szerkeszteni (link). A programkönyvtárak lehetővé teszik, hogy az alkalmazás modulárisabb, gyorsabban újrafordítható és könnyebben frissíthető legyen. A programkönyvtárakat három típusba sorolhatjuk: statikus programkönyvtárak, megosztott programkönyvtárak és dinamikusan betölthető (DL) programkönyvtárak.

Ez a leírás először a statikus programkönyvtárakkal foglalkozik, melyeket a program futtatása előtt kell az alkalmazáshoz szerkeszteni. Ezt követően foglalkozik a megosztott (shared) programkönyvtárakkal, amelyek a program indulásakor töltődnek be, és több program között megoszthatóak. Végül pedig a dinamikusan betölthető (DL) programkönyvtárakról lesz szó, amiket a programvégrehajtás alatt tölthetünk be. A DL programkönyvtárak nem igazán térnek el formátumban a másik két programkönyvtár-típustól (mind statikus, mind megosztott programkönyvtárak lehetnek DL programkönyvtárak), a különbség abból adódik, hogyan használják a programozók a DL programkönyvtárakat. A HOGYAN egy fejezetnyi példával és egy fejezetnyi hivatkozással zárul.

Minden programozónak, aki programkönyvtárakat fejleszt elvileg megosztott programkönyvtárakat kellene készítenie azért, hogy lehetővé tegye a felhasználóknak a programkönyvtárak alkalmazástól független frissítését. A dinamikusan betölthető (DL) programkönyvtárak hasznosak, de kicsivel több munkát igényel a használatuk, és sok programnak nincs szüksége arra a rugalmasságra amit nyújtanak. Ezzel szemben a statikus programkönyvtárak sokkal körülményesebbé teszik a frissítést. Ezért ritkán ajánlott a használatuk. Ezzel együtt mindegyik programkönyvtár-típusnak van előnye, mely előnyöket egy-egy fejezetben foglaljuk össze a későbbiekben. A dinamikusan betölthető (DL) programkönyvtárakat használó C++ fejlesztőknek a "C++ dlopen mini-HOGYAN" is ajánlott olvasmány.

Elég szerencsétlen, hogy jó néhányan a dinamikusan *szerkesztett* (linked) programkönyvtárak (DLL-ek) kifejezést használják a megosztott programkönyvtárakra. Mások ugyanezt a kifejezést bármely olyan programkönyvtárra használják, mely dinamikusan betölthető. Megint mások a DLL-t a programkönyvtárakra használják megkötések nélkül. Függetlenül attól, hogy te mit értesz alatta, ez a HOGYAN a DLL-ekkel foglalkozik Linuxon.

A HOGYAN csak a ELF (Executable and Linking Format) formátumú futtatható fájlokkal és programkönyvtárakkal foglalkozik. Ezt a formátumot használja a legtöbb Linux terjesztés. A GNU gcc eszközkészlet ettől eltérő formátumokat is képes kezelni, például a legtöbb Linux terjesztés még mindig használja az elavult a.out formátumot. Ugyanakkor ez a formátum kívül esik ennek a HOGYANnak a témakörén.

Ha hordozható alkalmazás szeretnél készíteni, akkor megfontolandó a GNU libtool (http://www.gnu.org/software/libtool/libtool.html) használata. Ebben az esetben a programkönyvtárakat ezzel az eszközzel készíted és telepíted, a linuxos eszközök közvetlen használata helyett. A GNU libtool egy általános programkönyvtár-készítést és telepítést támogató szkript-készlet, ami konzisztens és hordozható felülettel rejti el a megosztott programkönyvtárak használatának bonyolultságát. Linuxon a GNU libtool azokra az eszközökre és egyezményekre épül közvetlenül, melyeket ez a HOGYAN tárgyal. Számtalan hordozhatóságot biztosító illesztőfelület (wrapper) létezik dinamikusan betölthető programkönyvtárakhoz is. A GNU libtool tartalmaz egy ilyen illesztőfelület programkönyvtárat "libltdl" néven. Egy másik alternatíva lehet a glib programkönyvtár használata (nem összekeverendő a glibc-vel), ami hordozható támogatást nyújt a dinamikusan betölthető modulokhoz. További információt a http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html honlapon találsz. Még egyszer: Linuxon ez a funkció az ebben a HOGYANban leírt módszerekkel is megvalósítható. Ha jelenleg Linuxon fejlesztesz vagy hibát keresel, akkor valószínűleg az ebben a HOGYANban található információkra lesz szükséged.

A http://www.dwheeler.com/program-library webhely a központi elérési helye a HOGYANnak (angol verzió), az elkészítésében közreműködött a The Linux Documentation Project (http://www.tldp.org). A szerzői jogokat David

1.1. Magyar fordítás

A magyar fordítást Szalai Ferenc Attila (mailto:szferi[kukac]angel.elte[pont]hu) készítette (2003.12.11). A lektorálást Daczi László (mailto:dacas@freemail.hu_NO_SPAM) végezte el (2004.03.01). A dokumentum legfrissebb változata megtalálható a Magyar Linux Dokumentációs Projekt (http://tldp.fsf.hu/index.html) honlapján.

2. Statikus programkönyvtárak

A statikus programkönyvtárak közönséges tárgykód-fájlok gyűjteményei. Megállapodás szerint a statikus programkönyvtárak ".a" kiterjesztéssel végződnek. Egy ilyen gyűjtemény az ar (archiver) programmal készíthető. A statikus programkönyvtárakat ma már nem használják olyan gyakran, mint régebben, figyelembe véve a megosztott programkönyvtárak előnyeit (lásd lejjebb). Néha azonban, még mindig készítenek ilyeneket. Történelmileg ők jelentek meg először, és egyszerűbb is elmagyarázni működésüket.

A statikus programkönyvtárak lehetővé teszik a felhasználóknak, hogy újrafordítás nélkül szerkesszék őket a programokhoz, ezzel lerövidítve a fordítási időt. Megjegyezzük, hogy a mai gyors fordítók mellett a újrafordítási idő kevésbé fontos szempont, így ez ma már nem olyan erős érv, mint régebben. A statikus programkönyvtárak gyakran hasznosak azoknak a fejlesztőknek, akik lehetővé akarják tenni a programozók számára, hogy szerkesszék a programjukat a programkönyvtárukhoz, de nem akarják a programozók rendelkezésére bocsátani a programkönyvtár forrását. (Ez előnyös lehet a programkönyvtár forgalmazójának, de nyilvánvalóan hátrányos annak a programozónak, aki azt használni szeretné). Elméletileg a programhoz csatolt statikus ELF programkönyvtárban lévő kód kissé gyorsabban (1-5%-al) futhat, mint a megosztott vagy dinamikusan betölthető programkönyvtárban lévő. Gyakorlatilag ez ritkán ad okot a statikus programkönyvtárak használatára, az egyéb zavaró tényezők miatt.

A statikus programkönyvtár készítéséhez, vagy a már meglévő program könyvárhoz új tárgykód-fájlok hozzáadásához az alábbi utasítást használjuk:

```
ar rcs my_library.a file1.o file2.o
```

Ez az egyszerű utasítás hozzáadja a file1.o és file2.o tárgykód-fájlokat a my_library.a statikus programkönyvtárhoz. Létrehozza a my_library.a fájlt, ha az nem létezett. További információkat a statikus programkönyvtárak készítéséről az ar(1)-ben találsz.

Ha elkészítettél egy statikus programkönyvtárat nyilván használni is akarod. Úgy használhatod a statikus programkönyvtárat, hogy hivatkozol rá fordítási és szerkesztési folyamatnak abban a fázisában, amikor a program futtatható fájlja készül. Ha gcc(1)-et használsz a futtatható fájl készítésére, akkor a -l opcióval írhatod elő használni kívánt programkönyvtárak programhoz szerkesztését. További információt az info:gcc-ben találsz.

Légy óvatos a paraméterek sorrendjével, ha gcc-t használsz. Mivel a -l a szerkesztő (linker) kapcsolója, így azt a fordítandó fájl neve UTÁN tedd. Ez egy kicsit különbözik a normális kapcsolómegadási szintaxistól. Ha a fájl elé teszed a -l opciót, akkor a csatolás tökéletesen hibás lesz és misztikus hibákat eredményez.

Használhatod a linker programot ld(1) közvetlenül is a -l és -L opciókkal, de a legtöbb esetben jobb a gcc(1)-t használni, mivel az ld(1) interfésze nagyobb valószínűséggel változik, mint a gcc fordítóé.

3. Megosztott programkönyvtárak

Megosztott programkönyvtárak olyan programkönyvtárak, amiket a program indulásakor tölt be. Ha a megosztott programkönyvtárak megfelelően vannak telepítve, az összes program az indulása után automatikusan új megosztott programkönyvtárat használ. Ez ennél egy kicsit rugalmasabb és bonyolultabb jelenleg, mert a linuxos megoldás megengedi azt is, hogy:

- frissítsd a programkönyvtáraidat és közben támogasd azokat a programokat is, amik a régebbi változatait használják azoknak a programkönyvtáraknak, amik visszafele nem kompatibilisek.
- felülbírálj specifikus programkönyvtárakat vagy függvényeket a programkönyvtárban különleges programok futtatásakor.
- mindezt a programok futása közben tegyed, miközben azok a már létező programkönyvtárakat használják.

3.1. Konvenciók

A fent leírt tulajdonságok megvalósításához a megosztott programkönyvtáraknak számos konvenciót és irányelvet követniük kell. Fontos megérteni a különbséget a programkönyvtárak nevei között, különösen a "so-név" és a "valódi név" között (és azok kapcsolatát). Azt is meg kell értened, hogy hova kell elhelyezni ezeket a programkönyvtárakat a fájlrendszerben.

3.1.1. Megosztott programkönyvtárak nevei

Minden megosztott programkönyvtárnak van egy speciális neve, amit "so-név"-nek hívnak. Az so-névnek van egy "lib" előtagja, ezt követi a programkönyvtár neve, majd egy ".so" tag majd egy pont, és a verziószám (egy speciális kivétel az alacsony szintű C programkönyvtárak, amik nem kezdődnek "lib"-el). A verziószám akkor növekedik, ha az interfész változik. A "teljes so-név" tartalmazza előtagként a könyvtár (directory) nevét, amiben a programkönyvtár található. Egy működő rendszeren a teljes so-név egyszerűen egy szimbolikus hivatkozás a megosztott programkönyvtár valódi nevére.

Minden megosztott programkönyvtárnak van "valódi neve" is, ami annak az fájlnak a neve, ami a jelenlegi programkönyvtár kódját tartalmazza. A valódi név az so-névhez ad egy pontot, a minor számot, egy újabb pont-ot és a kibocsátási számot (release number). Az utolsó pont és a kibocsátási szám opcionális. A minor szám és a kibocsátási szám arra való, hogy tudd, pontosan melyik változata van az adott programkönyvtárnak telepítve. Megjegyezzük, hogy ezek a számok nem feltétlenül egyeznek meg azzal, amilyen verziószámmal a programkönyvtár dokumentációjában hivatkoznak, habár az megkönnyítené a dolgokat.

Van továbbá egy név, amit a fordításnál használunk, amikor a programkönyvtárra hivatkozunk (ezt "csatolási név"-nek fogjuk hívni). Ez egyszerűen a so-név verzió nélkül.

A megosztott programkönyvtárak használatának kulcsa a neveik szétválasztásában rejlik. A programokban a szükséges megosztott programkönyvtárak so-neveire van csak szükség. Ennek megfelelően, amikor egy megosztott programkönyvtárat készítesz, mindössze egy programkönyvtárat készítesz egy speciális fájlnéven (részletes verzió információkkal). Amikor telepíted az új változatát a programkönyvtárnak, akkor a megfelelő könyvtárak egyikébe kell elhelyezned és az ldconfig(8) programot kell futtatnod. Az ldconfig megvizsgálja a létező fájlokat és elkészíti a so-neveket, mint szimbolikus hivatkozásokat a valódi nevekre, ezzel együtt beállítja az /etc/ld.so.cache gyorstár-fájlt is.

Az ldconfig nem állítja be a szerkesztési neveket, általában ez megtörtént már a programkönyvtár telepítése során, a szerkesztési név egyszerűen egy szimbolikus hivatkozás a "legújabb" so-névre vagy a legújabb valódi névre. Az tanácsolnám legyen a csatolási név egy szimbolikus hivatkozás az so-névre, mivel a legtöbb esetben a frissített programkönyvtárat szeretnéd automatikusan használni, mikor csatolod azt a programodhoz. Megkérdeztem H. J,. Lu-t, miért nem állítja be automatikusan az ldconfig a szerkesztési neveket. A magyarázata alapvetően az volt, hogy talán a programkönyvtár legutóbbi verzióját szeretnéd futtatni a kóddal, de az is lehet, hogy egy *fejlesztői* változatra szeretnél hivatkozást egy régi - talán inkompatibilis - programkönyvtár helyett. Ezért az ldconfig nem kísérli meg kitalálni, hogy mit akarsz a programhoz csatolni, így a telepítőnek kell meghatározni azt, és módosítani a szimbolikus hivatkozást arra a programkönyvtárra, amit a szerkesztő használni fog.

Így az `/usr/lib/libreadline.so.3` egy teljes so-név, amit az ldconfig készített, például mint szimbolikus hivatkozást a `/usr/lib/libreadline.so.3.0` fájlra. Kellene lennie egy `/usr/lib/libreadline.so` szerkesztési névnek is, ami egy szimbolikus hivatkozás lehet a `/usr/lib/libreadline.so.3` fájlra.

3.1.2. Elhelyezés a fájlrendszerben

A megosztott programkönyvtárakat el kell helyezni valahol a fájlrendszerben. A legtöbb nyílt forrású szoftver a GNU szabványok (standards) követésére törekszik. További információt az info:standards#Directory_Variables info dokumentumban találhatsz. A GNU szabványok azt ajánlják, hogy alapértelmezésben minden programkönyvtárat az /usr/local/lib könyvtárba telepítsünk a forráskód közzétételekor (és minden parancsnak az /usr/local/bin könyvtárba kellene kerülnie). Ezek a szabványok konvenciókat is meghatároznak arra vonatkozólag, hogyan bíráljuk felül ezt az alapértelmezett beállítást a telepítési eljárás során.

A Filesystem Hierarchy Standard (FHS) meghatározza mi hol legyen egy terjesztésben (lásd: http://www.pathname.com/fhs). Az FHS szerint a legtöbb programkönyvtárat az /usr/lib könyvtárba kell telepíteni, de azokat amik az elinduláshoz szükségesek a /lib könyvtárban kellene lenniük, és azok a könyvtárak, melyek nem részei a rendszernek azokat kellene a /usr/local/lib könyvtárba rakni.

Nincs igazán ellentmondás a két dokumentum között. A GNU szabványok a fejlesztőknek, míg az FHS a terjesztést összeállítóknak (akik szelektíven felülbírálják a forrás alapértelmezett beállításait, rendszerint a rendszer csomagkezelőjével) szóló ajánlások. Gyakorlatilag ez szépen működik: a "legújabb" (valószínűleg hibás!) forrást, amit letöltesz, automatikusan a "local" könyvtárba (`/usr/local`) telepíti magát, azoknál a kódoknál, amik "megértek" a csomag kezelők magától érthetődben felülbírálják az alapértelmezett beállításokat, és elhelyezik a kódot az alapértelmezett helyére a terjesztésben. Megjegyezzük, hogy ha a programkönyvtárad meghív olyan programokat, amik csak a programkönyvtárak által hívhatóak, akkor ezeket a programokat a /usr/local/libexec (ami /usr/libexec a Linux terjesztésekben) kell elhelyezni. A Red Hat alapú rendszerek egy sajátossága, hogy az /usr/local/lib könyvtár nincs az alapértelmezett programkönyvtár-keresési útvonalban (lásd a megjegyzéseket az /etc/ld.so.conf fájlról lejjebb). Egy másik szokványos programkönyvtár-hely az /usr/X11R6/lib, ahova az X-winodows rendszerrel kapcsolatos programkönyvtárakat szokás elhelyezni. Megjegyezzük, hogy a /lib/security a PAM modulok által használt hely, de ezek rendszerint DL programkönyvtárak (lásd lejjebb).

3.2. Hogyan használjuk a programkönyvtárakat?

GNU glibc alapú rendszereken - ide tartozik az összes Linux rendszer - egy ELF bináris futtatható fájl elindítása automatikusan magával vonja a programbetöltő elindítását. Linux rendszereken ennek a betöltőnek a neve /lib/ld-linux.so.X (ahol X a verzió számot jelöli). Ez a betöltő keresi meg és tölti be az összes program által használt megosztott programkönyvtárat.

Azoknak az elérési utaknak a listája, amiben a betöltő a programkönyvtárakat keresi, az /etc/ld.so.conf fájlban található. Több Red Hat alapú terjesztésben a /usr/local/lib nem található meg ebben a fájlban. Ezt én hibának tekintem és az /usr/local/lib hozzáadást az /etc/ld.so.conf fájlhoz egy általános "javításnak" gondolom, ami minden Red Hat alapú rendszeren szükséges lehet.

Ha felül akarsz bírálni néhány eljárást egy programkönyvtárban, de meg akarod tartani az eredetit, akkor elhelyezheted a felülbírálandó programkönyvtárak nevét (.o fájlok) az /etc/ld.so.preload fájlban. Ezek az "előtöltött" programkönyvtárak elsőbbséget élveznek a standard programkönyvtárakkal szemben. Ez az előtöltő fájl általában átmenti foltozásra szolgál, és a terjesztések rendszerint nem tartalmaznak ilyeneket.

Nem igazán hatékony ezeket az elérési utakat megkeresni minden programindításakor. Ezért egy gyorstárazó módszert alkalmazunk. Az ldconfig(8) program alapértelmezésben az /etc/ld.co.conf fájlt olvasva beállítja a megfelelő szimbolikus hivatkozásokat a dinamikus hivatkozások könyvtáraiban (dynamic link directories) (így azok a konvenciókat fogják követni), és egy gyorstár-fájlt készít /etc/ld.so.cache néven. Ezt a gyorstárat használja aztán a többi program. Ez nagyon felgyorsítja a programkönyvtárak elérését. A következmény az, hogy az ldconfig parancsot minden esetben futtatni kell, ha új DLL-at adunk a rendszerhez vagy ha eltávolítjuk azt, vagy mikor átállítjuk a DLL programkönyvtárakat. Az ldconfig futtatás a leggyakoribb feladat, amit egy csomagkezelőnek el kell végeznie, mikor programkönyvtárat telepít. Indulásnál a dinamikus betöltő az /etc/ld.so.cache fájlt használja és utána tölti be a szükséges programkönyvtárakat.

Megjegyezzük, hogy a FreeBSD teljesen más fájlnevet használ ennek a gyorstárnak. FreeBSD-ben az ELF gyorstár a /var/run/ld-elf.so.hints, míg az a.out gyorstár a /var/run/ld.so.hints fájl. Az ldconfig(8) ezeket is frissíti, így ez az elnevezésbeli különbség csak néhány egzotikus esetben érdekes.

3.3. Környezeti változók

Számos környezeti változóval szabályozható az előbb bemutatott működés. Ezek a környezeti változók lehetővé teszik, hogy beavatkozzunk a programkönyvtárak betöltésének és használatának menetébe.

3.3.1. LD_LIBRARY_PATH

Átmenetileg helyettesíthetsz néhány programkönyvtárat bizonyos programok futtatásakor. Linuxon a LD_LIBRARY_PATH környezeti változó egy kettősponttal elválasztott listája azoknak az elérési utaknak, ahol a programkönyvtárakat keresi, a szokásos helyek előtt. Ez hasznos hibakereséskor, ha egy új vagy nem szokványos programkönyvtárat használunk valamilyen speciális célból. Az LD_PRELOAD környezeti változó egy kettősponttal elválasztott listája azoknak a megosztott programkönyvtáraknak, amelyek függvényei felüldefiniálják a standard programkönyvtárakét, mint ahogy azt az /etc/ld.so.preload teszi. Meg kell említenünk, hogy a LD_LIBRARY_PATH használható majdnem minden Unix-szerű rendszeren, de nem mindegyiken. Például ez a funkció elérhető HP-UX-on, de a környezeti változó neve SHLIB_PATH. AIX-on ezt a funkciót a LIBPATH nevű változóval érhetjük el (ugyanezzel a szintaxissal, kettősponttal elválasztott lista).

LD_LIBRARY_PATH kényelmes fejlesztésnél vagy tesztelésnél, de szükségtelen módosítania egy normál felhasználónak vagy akár egy telepítési eljárásnak normál esetben. A miértekről a "Why LD_LIBRARY_PATH is Bad" című írásban, a http://www.visi.com/~barr/ldpath.html honlapon olvashatsz. Ezek ellenére használható fejlesztéskor vagy teszteléskor, vagy ha egy probléma behatárolása nem megy másképpen. Ha nem akarod beállítani a LD_LIBRARY_PATH környezeti változót, akkor Linuxon közvetlenül futtathatod a programbetöltőt, megfelelő argumentumokkal. Az alábbi példában a PATH-ot fogja használni az LD_LIBRARY_PATH környezeti változó tartalma helyett, így futtatja a megadott fájlt.

```
/lib/ld-linux.so.2 --library-path PATH EXECUTABLE
```

Az ld-linux.so argumentum nélkül futtatva további információkat add arról, hogyan használhatod, de még egyszer hangsúlyozom, hogy normál körülmények között ne használd ezt, kizárólag hibakeresésre.

3.3.2. LD_DEBUG

Másik hasznos környezeti változó az LD_DEBUG, a GNU C betöltőben. Ez a változó arra kényszeríti a dl* függvényeket, hogy bőséges információt adjanak arról, hogy mit csinálnak. Például:

```
export LD_DEBUG=files
command_to_run
```

megjeleníti a fájlok és programkönyvtárak feldolgozását, mikor a programkönyvtárakat kezeli. Megjeleníti, milyen függőségeket talált a rendszer, és milyen so-k lesznek betöltve, milyen sorrendben. Az LD_DEBUG-t "bindings"-re állítva információkat kapunk a szimbólum kötéséről. A "libs" beállítás a programkönyvtárak keresési útvonalát mutatja. A "versions" beállítás pedig a verzió függőségeket jeleníti meg.

Az LD_DEBUG-ot "help"-re állítása után, ha megpróbálunk futtatni egy programot, a rendszer megjeleníti a lehetséges beállításokat (a programot nem futtatja le). Még egyszer megjegyezzük, hogy az LD_DEBUG-ot nem normál használatra tervezték, de nagyon hasznos lehet hibakeresésnél és tesztelésnél.

3.3.3. További környezeti változók

Számos más környezeti változó is van, ami a betöltési eljárást szabályozza. Ezek LD_ vagy RTLD_ előtagokkal kezdődnek. A legtöbb alacsony szintű hibakeresést tesz lehetővé a betöltési eljárásban, vagy speciális tulajdonságok megvalósítását teszik lehetővé. A legtöbb nem túl jól dokumentált. Ha szükséged van rájuk legjobb, ha elolvasod a betöltő forráskódját, ha többet akarsz megtudni ezekről a környezeti változókról.

A felhasználói beavatkozás engedélyezése a dinamikusan csatolt programkönyvtárak betöltésénél katasztrofális eredményre vezethet, setuid/setgid-es programok esetén. Ezért a GNU betöltőben (ami betölti a program többi részét annak indulásakor), ha a program setuid vagy setgid beállításokkal rendelkezik, akkor az eddig említett és a többi hasonló környezeti változó vagy figyelmen kívül marad, vagy hatása erősen korlátozva lesz. Ha az uid és az euid, vagy ha a gid és a egid különbözik, a betöltő feltételezi, hogy a program setuid vagy setgid-es (vagy annak gyermeke), ezért jelentősen csökkenti a csatolás kontrollálásának lehetőségét környezeti változókkal. Ha a GNU glibc programkönyvtár forráskódját olvasod, láthatod ezt, különösen az elf/rtlf.c és a sysdeps/generic/dl-sysdep.c fájlokat olvasva. Ez azt jelenti, ha eléred, hogy a uid és a euid, valamit a gid és egid megegyezzenek ezek a változók kifejtik teljes hatásukat, mikor a programot futtatod. Más Unix-szerű rendszerek hasonló okból kifolyólag, de máshogy kezelik ezt a problémát: a setuid és setgid-es programot indokolatlanul nem befolyásolhatjuk környezeti változókkal.

3.4. Megosztott programkönyvtárak készítése

Megosztott programkönyvtárat készíteni könnyű. Először egy tárgykód-fájlt kell készítenünk, amit a megosztott programkönyvtárba fogunk pakolni, a gcc -fPIC vagy -fpic kapcsolóinak segítségével. A -fPIC és -fpic opciók engedélyezik a "pozíció független kód" (position independent code) generálását, ami szükséges a megosztott programkönyvtárak készítéséhez (a különbségeket lásd alább). Az so-nevet a -Wl kapcsolóval adhatod meg a gcc-nek. A -Wl opció a szerkesztőnek (linker) szól (ebben az esetben a -soname linker opció) - a vesszők a -Wl után nem helyesírási hibák, és nem szabad szóközöket tenni közéjük. A megosztott programkönyvtár készítéséhez az alábbi formátumot használd:

```
gcc -shared -Wl,-soname,your_soname \
    -o library_name file_list library_list
```

Íme egy példa, ami két tárgykód-fájlt készít (a.o és b.o) aztán ezekből egy megosztott programkönyvtárat. Megjegyezzük, hogy ez a fordítás (compile) tartalmazza a hibakeresési (debug) információkat és a figyelmeztetések (warnings) generálását is, amik nem feltétlenül szükségesek a megosztott programkönyvtárak készítéséhez. A fordítás tárgykód-fájlokat állít elő (-c használata), és tartalmazza a szükséges -fPIC kapcsolót:

```
gcc -fPIC -g -c -Wall a.c
gcc -fPIC -g -c -Wall b.c
gcc -shared -Wl,-soname,libmystuff.so.1 \
    -o libmystuff.so.1.0.1 a.o b.o -lc
```

Íme néhány jó tanács:

- Ne használd a strip parancsot a keletkezett programkönyvtárra. Továbbá ne használd a -fomit-frame-pointer fordítási kapcsolót, hacsak nincs rá igazán szükséged. Az elkészült programkönyvtár működni fog ezekkel is, de a hibakeresőket használhatatlanná teszik.
- Használd a -fPIC vagy a -fpic kapcsolót a kód generáláskor. Mind a -fPIC, mind a -fpic kapcsolókkal készített kód célplatform-függő. A -fPIC választás mindig működik, de nagyobb kódot generálhat, mint a -fpic (könnyű megjegyezni: PIC nagy betűs, tehát nagyobb kódot generál) A -fpic opció használatával általában kisebb és gyorsabb kódot kapunk, de lesznek platformfüggő korlátozások, és számos globálisan látható szimbólum. A szerkesztő (linker) jelzi, hogy az így létrehozott tárgykód alkalmas-e megosztott programkönyvtár készítésére. Minden esetben, mikor bizonytalan vagy -fPIC kapcsolót válaszd, mert az mindig működik.
- Néhány esetben a "-Wl,-export-dynamic" gcc kapcsolók is szükségesek lehetnek a tárgykód-fájl létrehozásához. Normális esetben a dinamikus szimbólum táblázat csak azokat a szimbólumokat tartalmazza, amiket a dinamikus tárgykód-fájl használ. Ez a kapcsoló (mikor ELF típusú fájlt készítünk) hozzáadja az összes szimbólumot a dinamikus szimbólum táblázathoz (lásd ld(1) több információért). Ezt a kapcsolót kell használnod, amikor vannak "visszafelé függőségek" (reverse dependences), úgy mint amikor egy DL programkönyvtárban fel ne oldott szimbólumok vannak, amiket a hagyomány szerint definiálni kell abban a programban, amelyik be szeretné tölteni ezeket a programkönyvtárakat. A "visszafelé függőség" működéséhez a főprogramnak dinamikusan elérhetvé kell tennie ezeket a szimbólumokat. Megjegyezzük, hogy használhatod a "-rdynamic" kapcsolót a "-Wl,export-dynamic" helyett, ha csak Linux rendszeren dolgozol, de az ELF dokumentációja szerint az "-rdynamic" kapcsoló nem mindig működik gcc-nél nem Linux-alapú rendszereken.

Fejlesztés közben az egyik tipikus probléma annak a programkönyvtárnak a módosítása, amit más programok is használnak -- és nem szeretnéd, hogy a többi program is a "fejlesztési" programkönyvtárat használja, csak egy kiszemelt alkalmazást szeretnél tesztelni vele. Az ld "rpath" szerkesztési (link) opcióját kell használnod ebben az esetben, ami szerkesztési időben meghatározza a futásidejű programkönyvtár (runtime library) keresési útvonalát egy adott programra. A gcc-t az rpath opcióval az alábbi módon használhatod:

```
 -Wl,-rpath,$(DEFAULT_LIB_INSTALL_PATH)
```

Ha ezt az opciót használod amikor elkészíted (build) a programkönyvtár kliens programját, akkor nem kell azzal vacakolnod, hogy a LD_LIBRARY_PATH-al elkerüld a összeütközéseket vagy, hogy más módszerekkel elrejtsd a programkönyvtárat.

3.5. Megosztott programkönyvtárak telepítése és használata

Ha már készítettél megosztott programkönyvtárat, nyilván használni is akarod. Az egyik lehetőség, hogy egyszerűen a szabványos könyvtárak egyikébe másolod a programkönyvtárat (pl. /usr/lib) és lefuttatod a ldconfig(8)-ot.

Először is készítened kell egy megosztott programkönyvtárat valahol. Aztán be kell állítanod a szükséges szimbolikus hivatkozásokat (symbolic links), különösen a so-névről a valódi névre (akárcsak a verziómentes so-névről mutatót - ami a so-név ami ".so"-val végződik - azoknak a felhasználóknak, akik nem határoznak meg verziót egyáltalán). Az egyszerűbb megoldás, hogy az alábbi parancsot futtatod:

```
ldconfig -n directory_with_shared_libraries
```

Végül, amikor fordítod a programodat, meg kell mondanod a szerkesztőnek azokat a statikus és megosztott programkönyvtárakat, amiket használni akarsz. Erre használd a -l és -L kapcsolókat.

Ha nem tudod, vagy nem akarod telepíteni a programkönyvtáradat egy standard helyre (például nincs jogod módosítani az /usr/lib könyvtárat), akkor más megoldást kell választanod. Ebben az esetben telepítened kell a programkönyvtárat valahova, aztán elég információt kell adnod a programodnak, hogy megtalálja a programkönyvtárat. Többféle mód is létezik erre. Használhatod a gcc -L kapcsolóját egyszerűbb esetekben. Használhatod a "rpath"-os megoldást (lásd feljebb), különösen, ha csak egy speciális program használja a programkönyvtárat, ami a "nem-standard" helyen van. Használhatsz környezeti változókat is, hogy kézben tartsd a dolgokat. Az LD_LIBRARY_PATH környezeti változó használhatod, ami egy kettősponttal elválasztott listája azoknak az elérési utaknak, ahol a megosztott programkönyvtárakat keressük, mielőtt a szokásos helyeken próbálkoznánk. Ha bash-t használsz indíthatod a my_program-ot így:

```
LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH  my_program
```

Ha csak néhány kiválasztott függvényt akarsz módosítani, akkor megteheted, hogy készítesz egy módosító tárgykód-fájlt, és beállítod a LD_PRELOAD környezeti változót. A függvények ebben a tárgykódban csak azokat a függvényeket fogják felülírni, amik a programkönyvtárban szerepelnek (a többit nem változtatja meg).

Rendszerint nyugodtan frissítheted a programkönyvtárakat, ha API változás volt a programkönyvtár készítője feltételezhetően megváltoztatta a so-nevet. Ezért lehet több programkönyvtár egyszerre egy rendszeren, a megfelelő lesz kiválasztva mindegyik programhoz. Ha a program mégis összeomlik, a programkönyvtár frissítésének hatására, ráveheted, hogy használja a régebbi programkönyvtár-verziót. Ehhez másold a régi programkönyvtárat vissza a rendszerre valahova. Változtasd meg a program nevét (legyen a régi név ".orig" kiterjesztéssel), majd készíts egy kis indító-szkriptet, ami visszaállítja a régi programkönyvtárat a programnak. A régi programkönyvtárat elhelyezheted egy saját speciális helyre, ha akarod, használhatod a számozási konvenciót, hogy több verzió is lehessen ugyanabban a könyvtárban. Íme egy minta indító-szkript:

```
#!/bin/sh
export LD_LIBRARY_PATH=/usr/local/my_lib:$LD_LIBRARY_PATH
exec /usr/bin/my_program.orig $*
```

Kérlek ne használd ezt, amikor a saját programodat készíted. Próbálj meggyőződni arról, hogy a programkönyvtáraid vagy visszamenőlegesen kompatibilisek, vagy növelted a verzió számot a so-névben minden esetben, ha nem kompatibilis változást végeztél. Ez csak egy vészmegoldás a legrosszabb esetekre.

Egy program által használt megosztott programkönyvtárak listáját az ldd(1) programmal kaphatod meg. Tehát például, ha az ls program által használt megosztott programkönyvtárakat szeretnéd látni, írd a következőt:

```
ldd /bin/ls
```

Megkapod azoknak a so-nevek listáját, amiktől a program függ, a könyvtárral (directory) együtt, amiben azok a nevek feloldhatóak. Gyakorlatilag minden esetben legalább két függőséged lesz:

- /lib/ld-linux.so.N (ahol N 1 vagy több, rendszerint legalább 2). Ez a programkönyvtár ami betölti az összes többit.
- libc.so.N (ahol N 6 vagy több). Ez a C programkönyvtár. Minden más nyelv is igyekszik a C programkönyvtárat használni (ahelyett, hogy sajátot valósítana meg), így a legtöbb program legalább ezt az egyet használja.

Vigyázat: *ne* futtasd az ldd-t olyan programra, amiben nem bízol. Ahogy az világosan le van írva az ldd(1) kézikönyvében, az ldd úgy működik (a legtöbb esetben), hogy beállít egy speciális környezeti változót (ELF tárgykódok esetén az LD_TRACE_LOADED_OBJECTS-et) és futtatja a programot. Lehetséges egy megbízhatatlan program számára, hogy rávegye az ldd felhasználót mesterséges kód futtatására (ahelyett, hogy egyszerűen megmutatná az ldd információt). Tehát a biztonság kedvéért ne használd az ldd-t olyan programra, amiben nem bízol meg.

3.6. Inkompatibilis programkönyvtárak

Abban az esetben, ha egy programkönyvtár új változata binárisan inkompatibilis a régivel, akkor a so-nevet meg kell változtatni. C-ben négy alapvető oka lehet annak, hogy egy programkönyvtár inkompatibilissé válik:

1. A függvény viselkedése megváltozik, nem egyezik meg az eredeti specifikációval.
2. Nyilvános adatelemek megváltoztak (kivétel: hozzáadott opcionális elemek a struktúrák végén nem okoznak problémát, ha ezeket a struktúrákat csak a programkönyvtár foglalja le)
3. Nyilvános függvényt eltávolítottak.
4. A nyilvános függvény interfésze megváltozott.

Ha ezeket el tudod kerülni, akkor a programkönyvtáraid binárisan kompatibilisek lesznek. Más szóval az alkalmazás bináris interfésze (Application Binary Interface - ABI) kompatibilis maradhat, ha a fenti típusú változtatásokat elkerülöd. Például, hozz létre új függvényeket, de ne töröld a régieket. Hozzáadhatsz új elemeket a struktúrákhoz, de csak akkor, ha meggyőződtél róla, hogy a régi programok nem lesznek érzékenyek erre a változásra. Csak a struktúra végére adj új elemeket, és csak a programkönyvtár (és nem az alkalmazás) által elfoglalt struktúrákban teheted ezt meg. Az új elemek legyenek opcionálisak (vagy a programkönyvtár töltse ki őket) stb. Figyelem: valószínűleg nem tudod kiterjeszteni a struktúráidat, ha a felhasználók tömbökben használják azokat.

C++ (illetve egyéb fordítási időben használatos sablonokat (template) vagy "compiled dispatched" eljárásokat támogató nyelvek) esetén a helyzet kicsit trükkösebb. A fenti megkötéseket mind figyelembe kell venni, és meg sok minden mást is. Ennek oka, hogy néhány információ rejtve (under the covers) került megvalósításra a lefordított kódban. Ennek eredménye nem nyilvánvaló, ha nem tudod, hogy tipikusan, hogy szokták a C++-t megvalósítani. Szigorúan véve ezek nem "új" dolgok, csak arról van szó, hogy C++ használhat adatstruktúrákat úgy, hogy az a meglepetés erejével hathat. Az alábbi egy - a Troll Tech’s Technical FAQ (http://www.trolltech.com/developer/faq/tech.html#bincomp) alapján összeállított (feltehetőleg hiányos) listája azoknak a dolgoknak, amit nem tehetsz meg C++-ban, ha meg akarod tartani a kompatibilitást.

1. virtuális függvények újra megvalósításának hozzáadása, (hacsak nem vagy benne biztos, hogy a régi programok az eredeti megvalósítást fogják használni). Mert a fordító már fordítási időben (nem csatolási (link) időben) kiértékeli a SuperClass::virtualFunction() függvényhívást.
2. virtuális tagfüggvény hozzáadása vagy törlése, mert ez megváltoztathatja a minden alosztály vtbl-jének méretét és szerkezetét.
3. bármilyen olyan adattag típusának megváltoztatása vagy törlése, amit inline tagfüggvények érnek el.
4. osztályhierarchia megváltoztatása, kivéve az új levelek hozzáadását.
5. privát adattag hozzáadása vagy elvétele, mert ez megváltoztatja a méretét és szerkezetét minden alosztálynak.
6. public vagy protected tagsági függvények eltávolítása, hacsak nem inline típusúak.
7. public vagy protected tagfüggvény inline típusúvá tenni.
8. módosítani a inline típusú függvények működését, kivéve ha régi változat továbbra is működik.
9. a tagsági függvények hozzáférési jogának (public, protected vagy private) megváltoztatása egy hordozható programban, mert néhány fordító hozzáadja a hozzáférési jogot a függvénynévhez.

Ez a hosszú lista is jól mutatja, hogy a C++ programkönyvtár fejlesztőknek bizonyos esetekben sokkal jobban meg kell tervezniük a dolgokat, hogy megtartsák a bináris kompatibilitást. Szerencsére Unix-szerű rendszereken (a Linuxot is beleértve) egy programkönyvtárad több verziója lehet egyszerre betöltve. Így amíg van elég lemezterület, a felhasználók futtathatnak "régi" programokat, amik régi programkönyvtárakat használnak.

4. Dinamikusan betölthető (Dynamically Loaded; DL) programkönyvtárak

Dinamikusan betölthető (DL) programkönyvtárak olyan programkönyvtárak, amik nem a program indulásakor töltődnek be. Különösen hasznos modulok és plug-in-ek megvalósítására, mivel lehetővé teszi, hogy csak akkor töltsük be ezeket, ha szükséges. Például a Betölthető Hitelesítő Modul (Pluggable Authentication Modules; PAM) rendszer DL programkönyvtárakat használ, így lehetővé teszi a rendszergazdáknak, hogy beállítsák és átállítsák az hitelesítő/azonosító eljárásokat. A dinamikusan betölthető programkönyvtárak jól használhatóak továbbá olyan értelmezők megvalósítására, amik alkalmanként lefordítanak kódokat gépi kódra és a lefordított hatékony változatokat használják, mindezt leállás nélkül. Például ez az eljárás használható just-in-time fordítók vagy multi-user dungeon (MUD) megvalósítására.

Linuxon a DL programkönyvtárak jelenleg formailag semmilyen speciális tulajdonsággal nem rendelkeznek. Standard tárgykódként vagy megosztott programkönyvtárként készülnek, mint ahogy azt feljebb már bemutattuk. A fő különbség, hogy ezek a programkönyvtárak nem töltődnek be automatikusan a program csatolása vagy elindítása során. Ehelyett van egy API, aminek segítségével megnyithatjuk a programkönyvtárat, szimbólumokat kereshetünk benne, javíthatjuk a hibákat és bezárhatjuk a programkönyvtárat. C felhasználóknak a <dlfcn.h> header fájlt kell beszerkeszteni (include) ennek az API-nak a használatához.

Az interfész használata Linuxon alapvetően ugyanolyan mint Solaris-on, amit "dlopen()" API-nak fogok hívni. Ugyanakkor ezt az interfészt nem támogatja minden platform. HP-UX egy másik shl_load() mechanizmust használ és a Windows platform is egy teljesen más DLL interfészt használ. Ha a célod a széleskörű hordozhatóság, akkor valószínűleg valamilyen köztes programkönyvtárat kell használnod, ami elrejti a különbségeket a platformok között. Egy megoldás lehet a glib programkönyvtár, ami támogatja a modulok dinamikus betöltését. Ez az platformok

dinamikus betöltő rutinjait használja ahhoz, hogy egy hordozható interfészt valósítson meg ezekhez a függvényekhez. Többet tudhatsz meg a glib-ről a http://developer.gnome.org/doc/API/glib/glib-dynamic-loading-of-modules.html honlapon. Mivel a glib interfész jól dokumentált nem részletezem itt. Egy másik lehetőség a libltdl használata, ami része a GNU libtool (http://www.gnu.org/software/libtool/libtool.html)-nak. Ha többet akarsz ennél, akkor vess egy pillantást a CORBA Object Request Broker (ORB)-re. Ha még mindig érdekel, hogyan használhatod közvetlenül a Linux és Solaris interfészeket, akkor olvass tovább.

Azok a fejlesztők aki C++-t és dinamikusan betölthető (DL) programkönyvtárakat akarnak használni elolvashatják a "C++ dlopen mini-HOGYANt".

4.1. dlopen()

A dlopen(3) függvény megnyitja a programkönyvtárat és előkészíti használatra. C prototípusa az alábbi:

```
void * dlopen(const char *filename, int flag);
```

Ha filename "/"-el kezdődik (például egy abszolút elérési út), dlopen() így fogja használni ezt (nem fogja megkeresi a programkönyvtárat). Egyéb esetekben a dlopen() az alábbi sorrendben keresi a programkönyvtárakat:

1. Könyvtárak kettősponttal elválasztott listája a felhasználó LD_LIBRARY_PATH környezeti változójában.
2. Az /etc/ld.so.cache fájlban található programkönyvtárak listájában, amit az /etc/ld.so.conf-ból generáltunk.
3. /lib, aztán /usr/lib. Megjegyezzük, hogy ez pont a fordítottja a régi a.out betöltő viselkedésének. A régi a.out betöltő először az /usr/lib könyvtárban keresett aztán a /lib könyvtárban (lásd ld.so(8) man oldal). Normális körülmények között ez nem számít, a programkönyvtárnak az egyik vagy a másik könyvtárban kellene lennie (sohasem mindkettőben), mivel azonos névvel rendelkező különböző programkönyvtárak katasztrófát okoznak.

A dlopen()-ben a *`flag`* értéke RTLD_LAZY "akkor oldja fel a nem definiált szimbólumokat, amint egy kód a dinamikus programkönyvtárból futtatásra kerül", vagy RTLD_NOW "felold minden nem definiált szimbólumot, mielőtt a dlopen() visszatér és hibát ad vissza, ha ez sikertelen volt". RTLD_GLOBAL opcionálisan vagy kapcsolatban használható bármelyikkel a *`flag`*-ban, ami azt jelenti, hogy a programkönyvtárban definiált külső (external) szimbólumok elérhetőek lesznek a többi betöltött programkönyvtárban is. Hibakeresés közben valószínűleg az RTLD_NOW-t akarod majd használni. Az RTLD_LAZY könnyen misztikus hibákat okozhat, ha feloldhatatlan referenciáid vannak (unresolved references). Az RTLD_NOW esetén kicsit tovább tart a programkönyvtár betöltése (de felgyorsítja a keresést később). Ha ez felhasználói interfész problémát okozna, akkor RTLD_LAZY-re válthatsz.

Ha a programkönyvtárak függnek egymástól (pl.: X függ Y-tól), akkor először a függőségeket kell betöltened (a példában Y-t és aztán X-et).

A dlopen() visszatérési értéke egy "handle", amire úgy tekinthetsz, mint egy opaque érték, amit más DL programkönyvtárak használhatnak. A dlopen() NULL-al fog visszatérni, ha a betöltés sikertelen volt. Ezt ellenőrizned is kell. Ha ugyanazt a programkönyvtárat többször töltöd be dlopen()-el, az ugyanazt az fájlkezelőt (handle) fogja visszaadni.

Ha a programkönyvtár tartalmaz egy _init nevű public eljárást, akkor az abban lévő kód lefut, mielőtt a dlopen visszatér. Ezt a saját program inicializációs eljárásnak használhatod. Ugyanakkor a programkönyvtáraknak nem kötelező _init és _fini nevű eljárásokat tartalmazniuk. Ez a mechanizmus elavult és nem kívánatos működést eredményezhet. Helyette a programkönyvtáraknak __attribute__((constructor)) és __attribute__((destructor))

függvény attribútumokkal megjelölt eljárásokat kellene használniuk (feltéve, hogy gcc-t használsz). További részleteket a "Programkönyvtár konstruktor és destruktor függvények" (#init-and-cleanup) fejezetben olvashatsz erről.

4.2. dlerror()

Hibákat a dlerror() függvényhívással lehet kezelni. Ez visszatér az utolsó dlopen(), dlsym() vagy dlclose() függvényhívás okozta hibát leíró karaktersorozattal. Kellemetlen, hogy a dlerror() meghívása után a többi dlerror() függvényhívás NULL-al fog visszatárni mindaddig, amíg egy másik hiba nem keletkezik.

4.3. dlsym()

Nincs értelme betölteni egy DL programkönyvtárat, ha nem tudod használni. A DL programkönyvtár használatának alaprutinja a dlsym(3). Ez szimbólumokat keres a már megnyitott programkönyvtárakban. A függvénydefiníció így néz ki:

```
void * dlsym(void *handle, char *symbol);
```

A handle a dlopen által visszaadott érték, a symbol egy NIL-el végződő karaktersorozat. Az a jó, ha a dlsym eredményét nem tárolod void* mutatóban, ellenkező esetben minden alkalommal ki kell osztanod (cast). (és kevés információt adsz azoknak az embereknek, akik megpróbálják megérteni a programodat)

A dlsym() NULL-al tér vissza, ha a szimbólumot nem találta. Ha előre tudod, hogy a szimbólum értéke soha nem NULL vagy zero, akkor ez rendben van, de potenciális veszélyforrás minden más esetben. Ugyanis, ha kapsz egy NULL-t nem tudod eldönteni, hogy a szimbólumot nem találta a dlsym, vagy az értéke éppen NULL. A szokásos megoldás ilyenkor, hogy először meghívod a dlerror()-t (azért, hogy eltüntess, minden hibát ami létezik), aztán a dlsym()-et hívod a szimbólum keresésére, végül újra a dlerror()-t, hogy lásd történt-e hiba. A kódrészlet valahogy így nézhet ki:

```
dlerror(); /* clear error code */
s = (actual_type) dlsym(handle, symbol_being_searched_for);
if ((err = dlerror()) != NULL) {
 /* handle error, the symbol wasn't found */
} else {
 /* symbol found, its value is in s */
}
```

4.4. dlclose()

A dlopen() párja a dlclose(), ami bezárja a DL programkönyvtárat. A DL programkönyvtár kapcsolatszámlálókat (link counts) hoz létre a dinamikus fájlkezelőkhöz, így a dinamikus programkönyvtár mindaddig nem lesz felszabadítva, amíg a dlclose-t nem hívták meg annyiszor, ahányszor a dlopen-t. Így nem okoz problémát, ha ugyanaz a program ugyanazt a programkönyvtárat többször tölti be. Ha a programkönyvtár valóban felszabadul a régi programkönyvtárak esetén a _fini függvénye meghívódik (ha létezik). A _fini már egy elavult mechanizmus és nem ajánlatos a használata. Helyette a programkönyvtáraknak az __attribute__((constructor)) és __attribute__((destructor)) függvényattribútumukkal ellátott eljárásit kell használni. További részleteket a

"Programkönyvtár konstruktor és destruktor függvények" (#init-and-cleanup) fejezetben olvashatsz erről. Megjegyezzük, hogy a dlclose() 0-val tér vissza siker, és nem nullával hiba esetén. Néhány Linux kézikönyv oldal nem tesz említést erről.

4.5. DL programkönyvtár példa

Íme egy példa a dlopen(3) kézikönyvből. Ez a példa betölti a math programkönyvtárat és kiírja a 2.0 koszinuszát. Ellenőrzi a hibákat, minden lépésnél (ami melegen ajánlott):

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>

int main(int argc, char **argv) {
    void *handle;
    double (*cosine)(double);
    char *error;

    handle = dlopen ("/lib/libm.so.6", RTLD_LAZY);
    if (!handle) {
        fputs (dlerror(), stderr);
        exit(1);
    }

    cosine = dlsym(handle, "cos");
    if ((error = dlerror()) != NULL)  {
        fputs(error, stderr);
        exit(1);
    }

    printf ("%f\n", (*cosine)(2.0));
    dlclose(handle);
}
```

Ha ez a program a "foo.c" állományban van, akkor az alábbi paranccsal fordíthatod le:

```
gcc -o foo foo.c -ldl
```

5. Egyéb

5.1. nm utasítás

Az nm(1) utasítás megmutatja milyen szimbólumok találhatóak egy adott programkönyvtárban. Mind statikus, mind megosztott programkönyvtárakra működik. Egy adott programkönyvtárra az nm(1) a szimbólumok neveit, és minden

szimbólum típusát és értékét képes megmutatni. Azt is képes meghatározni, hogy a forráskódban hol volt a szimbólum definiálva (fájlnév és sor), ha ezt az információt tartalmazza a programkönyvtár (lásd a -l kapcsolót).

A szimbólum típusok kicsit több magyarázatot igényelnek. A típust egy karakter jelzi. A kisbetű azt jelenti, hogy a szimbólum lokális, míg a nagybetű azt, hogy globális (külső). A tipikus szimbólum típusok a következők: T (normál definíció a kód részben), D (inicializált adat szekció), B (nem inicializált adat szekció), U (nem definiált; a szimbólumot használja a programkönyvtár de nincs definiálva abban), és W (waek; ha más programkönyvtár szintén definiálja ezt a szimbólumot, és az a definíció elrejti azt).

Ha tudod a függvény nevét, de nem emlékszel, hogy melyik programkönyvtárban van definiálva, akkor a nm "-o" kapcsolóját (ami minden sor elé odarakja az fájlnevet) használhatod a grep-el együtt, hogy megtaláld a programkönyvtár nevét. Bourne héjban kereshetsz minden programkönyvtárban a /lib, /usr/lib, /usr/local/lib és azok alkönyvtáraiban a "cos"-ra az alábbiak szerint:

```
nm -o /lib/* /usr/lib/* /usr/lib/*/* \
      /usr/local/lib/* 2> /dev/null | grep 'cos$'
```

Sokkal több információt található az nm-ről a "info" dokumentumában (info:binutils#nm).

5.2. Programkönyvtár konstruktor és destruktor függvények

A programkönyvtárak megvalósíthatnak nyilvános inicializációs és cleanup függvényeket, a gcc __attribute__((constructor)) és __attribute__((destructor)) függvény attribútumait használva. További információt a gcc info oldalain találsz erről. A konstruktor eljárások a dlopen visszatérése előtt (vagy a main() előtt, ha a programkönyvtár betöltési időben kerül megnyitásra) hívódnak meg. A destruktor eljárások a dlclose visszatérése előtt (vagy a exit() vagy a main() befejezésekor, ha a programkönyvtár betöltési időben került megnyitásra) hívódnak meg. A C prototípusa ezeknek a függvényeknek a következő:

```
  void __attribute__ ((constructor)) my_init(void);
  void __attribute__ ((destructor)) my_fini(void);
```

A megosztott programkönyvtárakat tilos a gcc "-nostartfiles" vagy "-nostdlib" kapcsolóival fordítani. Ha ezeket használjuk a konstruktor/destruktor eljárások nem fognak lefutni (hacsak valami speciális nem történik).

5.2.1. Speciális függvények, _init és _fini (ELAVULT/VESZÉLYES)

Történelmileg létezik két speciális függvény a _init és a _fini, amik lehetővé teszik a konstruktorok és a destruktor vezérlését. Ugyanakkor ezek elavultak és használatuk megjósolhatatlan viselkedést eredményezhet. A programkönyvtáraidban szükségtelen ezeket használnod. Helyettük a fenti függvény argumentumokkal ellátott konstruktor és destruktor függvényeket használd.

Ha régi rendszeren kell dolgoznod, vagy olyan kóddal ami használja a _init vagy a _fini függvényeket, akkor itt található a működési leírásuk. Két speciális függvényt definiáltak a modulok inicializálására és a befejezésre: _init és a _fini. Ha a programkönyvtárban nyilvános "_init" függvény van definiálva, akkor az meghívódik, amikor a programkönyvtárat először megnyitjuk (dlopen()-el vagy egyszerűen megosztott programkönyvtárként). C programban ez egyszerűen azt jelenti, hogy definiálsz egy függvényt _init névvel. Létezik egy _fini nevű függvény, ami meghívódik mindannyiszor, ha a program befejezi a programkönyvtár használatát. (dlclose() meghívásával, amikor az a referencia számlót 0-ra állítja vagy a program normális kilépésekor). A C prototípusok:

```
  void _init(void);
  void _fini(void);
```

Ebben az esetben, ha az tárgykódot készítünk (".o") gcc-vel a "-nostartfiles"opciót meg kell adnunk. Ez megóvja a C fordítót attól, hogy a .so fájljal szemben a rendszerindítási programkönyvtárakat satolja a programhoz. Ellenkező esetben "multiple-definition" (többszörös definíció) hibát fogsz kapni. Megjegyezünk, hogy ez teljesen eltér attól az esettől, amikor modulokat a javasolt függvényattribútumos megoldással fordítasz. Köszönet Jim Mischel-nek és Tim Gentry-nek a javaslatárt, hogy kerüljön ide egy leírás a _init és _fini-ről, és hogy segítettek elkészíteni azt.

5.3. Megosztott programkönyvtárak, mint szkriptek

Jó dolog, hogy a GNU betöltő engedélyezi, hogy a programkönyvtárak szöveges fájlok is lehetnek, amiket egy speciális szkript nyelven kell írni a szokásos programkönyvtár-formátum helyett. Ez hasznos például programkönyvtárak közvetett összekapcsolása esetén. Példaként álljon itt a rendszerem `/usr/lib/libc.so` fájlja:

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
GROUP ( /lib/libc.so.6 /usr/lib/libc_nonshared.a )
```

További információt találsz erről a ld texinfo dokumentációjában a linker scripts szekcióban (ld parancs nyelv). Általános információ a info:ld#Options and info:ld#Commands, fejezetben. Az utasítások pedig a info:ld#Option Commands-ban.

5.4. Szimbólum verziók és verzió szkriptek

A külső függvényekre történő hivatkozások jellemzően egy szükségszerű bázishoz vannak kötve, amelyek közül nem mindegyik kötődik a külső függvényhez, az alkalmazás indulásakor. (Typically references to external functions are bound on an as-needed basis, and are not all bound when the application starts up.) Ha a megosztott programkönyvtár régi, az igényelt interfésze hiányozhat, amikor az alkalmazás megpróbálja használni azt. Ez azonnali és váratlan hibát okozhat.

A probléma megoldható a szimbólumok verzióval való megjelölésével, és a verzió-szkript csatolásával. Szimbólum verziók használatával a felhasználó figyelmeztetést kap, ha olyan programot indít el, amely túl régi programkönyvtárat akar használni. Többet tudhatsz meg az ld kézkönyvből, a http://www.gnu.org/manual/ld-2.9.1/html_node/ld_25.html honlapon.

5.5. GNU libtool

Ha olyan alkalmazást készítesz amit több rendszerre szeretnél használni, akkor javasolt a GNU libtool (http://www.gnu.org/software/libtool/libtool.html) használata a programkönyvtár fordításához és telepítéséhez. A GNU libtool egy általános programkönyvtár-készítést támogató szkript. A Libtool elrejti a megosztott programkönyvtárak bonyolultságát egy konzisztens és hordozható interfész mögé. A Libtool hordozható interfészt biztosít a tárgykód fájlok készítéséhez, a programkönyvtárak (statikus és megosztott), a futtatható fájlokhoz

csatolásához, a futtatható fájlok hibakereséséhez, a programkönyvtárak és futtatható fájlok telepítéséhez. Tartalmaz továbbá egy libltdl programkönyvtárat, ami egy hordozható csatolófelület a dinamikusan betölthető programkönyvtárakhoz. Többet tudhatsz meg a http://www.gnu.org/software/libtool/manual.html honlapon található dokumentációból.

5.6. Szimbólumok eltávolítása

A generált fájlokban található szimbólumok hasznosak hibakeresésnél, de sok helyet foglalnak. Ha helyre van szükséged, akkor csökkentheted a szimbólumok számát.

A legjobb eljárás először a tárgykódokat szimbólumokkal generálni és a tesztelést és hibakaresést ezekkel végezni, sokkal könnyebb így. Azután, mikor a program már alaposan tesztelve van a strip(1)-et használhatod a szimbólumok eltávolítására. A strip(1) utasítás sok lehetőséget ad arra, hogy meghatározd milyen szimbólumokat akarsz eltávolítani. További részletekért olvasd a dokumentációt.

Egy másik lehetőség a GNU ld "-S" és "-s" kapcsolóinak használata. Az "-S"'mellőzi a hibakereséshez szükséges szimbólumokat (de nem az összes szimbólumot). A "-s" pedig az összes szimbólum információt kihagyja a kimeneti fájlból. Használhatod ezeket a kapcsolókat a gcc-n keresztül is a "-Wl,-S" és "-Wl,-s" formában. Ha sosem akarsz szimbólumokat, akkor ezek a kapcsolók megfelelőek, használd őket. De ez a kevésbé rugalmas megoldás.

5.7. Nagyon kicsi futtatható fájlok

Egy igen hasznos leírást találhatsz a Whirlwind Tutorial on Creating Really Teensy ELF Executables for Linux (http://www.muppetlabs.com/~breadbox/software/tiny/teensy.html) honlapon. Ez leírja, hogyan készíthetsz igazán parányi futtatható fájlokat. Őszintén szólva a legtöbb ott található trükk nem használható normál esetben, de jól mutatják, hogy működik az ELF igazából.

5.8. C++ vs. C

Semmi akadálya annak, hogy C++ programból C programkönyvtári függvényeket hívj meg. A C++ kódban a C függvény extern "C"-nek kell definiálni. Ellenkező esetben a szerkesztő (linker) nem fogja megtalálni a C függvényt. A C++ fordító "szétszedi" a C++ függvények neveit (például típusazonosítás céljából), jelezni kell neki, hogy egy adott függvényt, mint C függvényt kell hívnia (és így ezt a szétszedést nem kell használni).

Ha olyan programkönyvtárat írsz amit C és C++-ból is hívni kell, akkor ajánlott a megfelelő header fájlba elhelyezd a 'extern "C"'utasítást azért, hogy ezt a tulajdonságot automatikusan biztosítsd a felhasználóknak. Ha ezt kombinálod a szokásos #ifndef-el a fájl elején, hogy elkerüld a header fájl újrafeldolgozását, akkor ez azt jelenti, hogy egy tipikus header fájl, ami C és C++-ban is használható (neve legyen mondjuk foobar.h) így néz ki:

```
/* Explain here what foobar does */

#ifndef FOOBAR_H
#define FOOBAR_H

#ifdef __cplusplus
extern "C" {
#endif

 ... header code for foobar goes here ...
```

```
#ifdef  __cplusplus
}
#endif
#endif
```

5.9. A C++ inicializáció felgyorsítása

A KDE fejlesztők jelezték, hogy nagy grafikus C++ alkalmazások indulása sokáig tart. Ez részben a sok újra allokációnak köszönhető. Létezik néhány megoldás a problémára. További információt Waldo Bastian: Making C++ ready for the desktop (http://www.suse.de/~bastian/Export/linking.txt) című írásában olvashatsz.

5.10. Linux Standard Base (LSB)

A Linux Standard Base (LSB) projekt célja, hogy olyan szabványokat dolgozzon ki és népszerűsítsen, amelyek növelik a kompatibilitást a Linux terjesztések között, és lehetővé teszik az alkalmazások futtatását minden ennek megfelelő Linux rendszeren. A projekt honlapja a http://www.linuxbase.org webhelyen érhető el.

Egy szép cikk jelent meg George Kraft IV (IBM Linux Technology Center senior szoftvermérnök) tollából 2002 októberében arról, hogyan fejlesszünk LSB kompatibilis alkalmazásokat: Developing LSB-certified applications: Five steps to binary-compatible Linux applications (http://www-106.ibm.com/developerworks/linux/library/l-lsb.html?t=gr,lnxw02=LSBapps). Természetesen a kódot úgy kell megírni, hogy egy standardizált réteget használjon, ha azt akarod, hogy a program hordozható legyen. Továbbá a LSB eszközöket biztosít a C/C++ az alkalmazás készítőknek, a programok LSB kompatibilitásának ellenőrzésére. Ezek az eszközök kihasználják a szerkesztő (linker) néhány tulajdonságát, és néhány speciális programkönyvtárat használnak a szükséges ellenőrzéseknek elvégzésére. Nyilvánvalóan telepítened kell ezeket az eszközöket, ha el akarod végezni ezeket az ellenőrzéseket. Az LSB honlapján megtalálhatók. Ezt követően egyszerűen a "lsbcc" kell használnod, mint C/C++ fordítót (az lsbcc belsőleg készít egy csatolási környezetet, ami jelezni fogja, ha bizonyos LSB szabályok sérülnek):

```
$ CC=lsbcc make myapplication
 (or)
$ CC=lsbcc ./configure; make myapplication
```

Az lsbappchk programot arra használhatod, hogy ellenőrizd a program valóban csak az LSB által standardizált függvényeket használja:

```
$ lsbappchk myapplication
```

Az LSB csomagolási útmutatót is követned kell (pl. használj RPM v3-at, használj LSB által meghatározott csomagneveket, és az add-on szoftvereket az /opt-ba kell telepítened alapértelmezetten). További információkat a cikkben és az LSB honlapján találsz.

6. További példák

Az alábbi példák mindhárom programkönyvtár típussal foglalkoznak (statikus, megosztott és dinamikus programkönyvtárak). A libhello.c fájl egy triviális programkönyvtár, a libhello.h header-rel. A demo_use.c fájl egy triviális program, ami a programkönyvtárat használja. Ezt követik a dokumentált szkriptek (script_static és script_dynamic), amik bemutatják, hogyan használhatod a programkönyvtárat megosztott illetve statikus programkönyvtárként. Ezután a demo_dynamic.c fájljal és a script_dynamic szkripttel megmutatjuk, hogyan használhatod a megosztott programkönyvtárat, mint dinamikusan betöltető programkönyvtárat.

6.1. libhello.c fájl

```
/* libhello.c - demonstrate library use. */

#include <stdio.h>

void hello(void) {
  printf("Hello, library world.\n");
}
```

6.2. libhello.h fájl

```
/* libhello.h - demonstrate library use. */


void hello(void);
```

6.3. demo_use.c fájl

```
/* demo_use.c -- demonstrate direct use of the "hello" routine */

#include "libhello.h"

int main(void) {
 hello();
 return 0;
}
```

6.4. script_static fájl

```
#!/bin/sh
# Static library demo

# Create static library’s object file, libhello-static.o.
# I’m using the name libhello-static to clearly
# differentiate the static library from the
# dynamic library examples, but you don’t need to use
# "-static" in the names of your
# object files or static libraries.

gcc -Wall -g -c -o libhello-static.o libhello.c

# Create static library.

ar rcs libhello-static.a libhello-static.o

# At this point we could just copy libhello-static.a
# somewhere else to use it.
# For demo purposes, we’ll just keep the library
# in the current directory.

# Compile demo_use program file.

gcc -Wall -g -c demo_use.c -o demo_use.o

# Create demo_use program; -L. causes "." to be searched during
# creation of the program.  Note that this command causes
# the relevant object file in libhello-static.a to be
# incorporated into file demo_use_static.

gcc -g -o demo_use_static demo_use.o -L. -lhello-static

# Execute the program.

./demo_use_static
```

6.5. script_shared fájl

```
#!/bin/sh
# Shared library demo

# Create shared library’s object file, libhello.o.

gcc -fPIC -Wall -g -c libhello.c

# Create shared library.
# Use -lc to link it against C library, since libhello
```

```
# depends on the C library.

gcc -g -shared -Wl,-soname,libhello.so.0 \
    -o libhello.so.0.0 libhello.o -lc

# At this point we could just copy libhello.so.0.0 into
# some directory, say /usr/local/lib.

# Now we need to call ldconfig to fix up the symbolic links.

# Set up the soname.  We could just execute:
# ln -sf libhello.so.0.0 libhello.so.0
# but let's let ldconfig figure it out.

/sbin/ldconfig -n .

# Set up the linker name.
# In a more sophisticated setting, we'd need to make
# sure that if there was an existing linker name,
# and if so, check if it should stay or not.

ln -sf libhello.so.0 libhello.so

# Compile demo_use program file.

gcc -Wall -g -c demo_use.c -o demo_use.o

# Create program demo_use.
# The -L. causes "." to be searched during creation
# of the program; note that this does NOT mean that "."
# will be searched when the program is executed.

gcc -g -o demo_use demo_use.o -L. -lhello

# Execute the program.  Note that we need to tell the program
# where the shared library is, using LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo_use
```

6.6. demo_dynamic.c fájl

```
/* demo_dynamic.c -- demonstrate dynamic loading and
   use of the "hello" routine */


/* Need dlfcn.h for the routines to
   dynamically load libraries */
#include <dlfcn.h>
```

```
#include <stdlib.h>
#include <stdio.h>

/* Note that we don’t have to include "libhello.h".
   However, we do need to specify something related;
   we need to specify a type that will hold the value
   we’re going to get from dlsym(). */

/* The type "simple_demo_function" describes a function that
   takes no arguments, and returns no value: */

typedef void (*simple_demo_function)(void);


int main(void) {
 const char *error;
 void *module;
 simple_demo_function demo_function;

 /* Load dynamically loaded library */
 module = dlopen("libhello.so", RTLD_LAZY);
 if (!module) {
   fprintf(stderr, "Couldn’t open libhello.so: %s\n",
           dlerror());
   exit(1);
 }

 /* Get symbol */
 dlerror();
 demo_function = dlsym(module, "hello");
 if ((error = dlerror())) {
   fprintf(stderr, "Couldn’t find hello: %s\n", error);
   exit(1);
 }

 /* Now call the function in the DL library */
 (*demo_function)();

 /* All done, close things cleanly */
 dlclose(module);
 return 0;
}
```

6.7. script_dynamic fájl

```
#!/bin/sh
# Dynamically loaded library demo

# Presume that libhello.so and friends have
```

```
# been created (see dynamic example).

# Compile demo_dynamic program file into an object file.

gcc -Wall -g -c demo_dynamic.c

# Create program demo_use.
# Note that we don’t have to tell it where to search for DL libraries,
# since the only special library this program uses won’t be
# loaded until after the program starts up.
# However, we DO need the option -ldl to include the library
# that loads the DL libraries.

gcc -g -o demo_dynamic demo_dynamic.o -ldl

# Execute the program.  Note that we need to tell the
# program where get the dynamically loaded library,
# using LD_LIBRARY_PATH.

LD_LIBRARY_PATH="." ./demo_dynamic
```

7. További információ

További hasznos információkat találsz a programkönyvtárakról a következő forrásokban:

- Daniel Barlow: "The GCC HOWTO". Ez a HOGYAN különösen a fordító (compiler) kapcsolókat tárgyalja, amivel programkönyvtárakat készíthetünk. Olyan információkat tartalmaz amivel itt nem foglalkoztunk és viszont. A HOGYAN a Linux Documantation Project oldalán keresztül érhető el: http://www.tldp.org.
- Tool Interface Standards (TIS) biztoság: "Executable and Linkable Format (ELF)" (ez jelenleg egy fejezete a Portable Formats Specification Version 1.1.-nek ugyanettől a bizottságtól). Ez az ELF formátumról tartalmaz nagy mennyiségű és részletes információt (ami nem Linux vagy GNU gcc specifikus). Lásd: ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz Ha az MIT-ről szeded le a fájlt, akkor kicsomagolás után egy "hps" állományt fogsz kapni. Csak töröld az első és utolsó sort, majd nevezd át "ps"-re és egy nyomtatható Postscript állományt kapsz a szokásos fájlnévvel.
- Hongjui Lu: "ELF: From the Programmer’s Perspective" . Linux és GNU specifikus információkat tartalmaz a ELF-ről, a ftp://tsx-11.mit.edu/pub/linux/packages/GCC/elf.ps.gz webhelyen érhető el.
- Az ld dokumentációja: "Using LD, the GNU Linker" írja le az ld-t messze a legrészletesebben. A http://www.gnu.org/manual/ld-2.9.1 webhelyen érhető el.

8. Copyright and License