D&O Linux Protokoll
Shared Libraries

Benedikt Horneber
Florian Forster
Florian Penzkofer

2004-01-07

1 Notwendigkeit

Die `libc` enthält alle `SYSCALL`s und muss daher in jedes Ausführbares Programm eingebunden werden. Durch den Umfang der `libc` (derzeit ca. 3 MB) ist es nicht praktikabel, die gesamte `libc` an alle Binaries anzuhängen.

2 Das `ar` Format

`ar`-Archive sind eine Sammlung bereits compilierter Funktionen, auf die beliebig zugegriffen werden kann. Die Datei `libc.a` andhält beispielsweise (unter Anderem) die Funktionen `printf.o`, `write.o`, `exit.o`, etc.

Daraus ergeben sich folgende Probleme:

- Viele Funktionen (zum Beispiel `printf`) sind nach wie vor redundant vorhanden.
- Eine Änderung an der Library würde das neu übersetzen aller Programme erfordern.

3 Das `so` Format

„**S**hared **O**bjects“ sind in sich konsistente Dateien, die zur Laufzeit in den Speicher geladen werden und müssen nicht zu jedem Programm hinzukopiert werden.[1]

3.1 Versionen

Da sich die Schnittstellen von Libraries mit der Zeit ändern (können) ist es sinnvoll, Versionsnummern zur verfügung zu stellen. So können Programme abhängigkeiten von bestimmten Versionen angeben und müssen nicht mit einem Programmabsturz feststellen, dass eine verwendete Version nicht die Richtige war.

[1] Das stimmt nicht ganz, siehe Abschnitt 3.5

Dabei ist ein durchdachter und konsistenter Ansatz zur Versionsnummern-Vergabe wichtig. Versionsnummern sind in aller Regel wie folgt aufgebaut:

$$< major > . < minor > . < bugfix >$$

major Die *major*-Nummer einer Library sollte sich genau dann ändern, wenn sich das Interface ändert.

minor Die *minor*-Nummer ändert sich bei Änderungen innerhalb der Library. Das Interface darf sich jedoch nicht ändern.

bugfix Die *bugfix*-Nummer ändert sich durch das Finden und Beheben von Bugs.

Am häufigsten wird folgende Namenskonvention verwendet:
`libc_`$< major > . < minor > . < bugfix >$`.so` (z.B.: `libc_2.3.3.so`)
Manchmal sieht man auch etwas in diese Richtung: `libc.so_2.3.3`

3.2 Auswahl einer Version

Beim linken eines Programms steht auch der Flag `-lc`, d.h. der Linker soll die `libc` beachten, aber welche Version? Lösungen wären:

- einfach immer die neueste Version nehmen, was aber evtl. Probleme machen könnte, da sich Schnittstellen geändert haben könnten,
- mit `-Lc` den Pfad direkt angeben
- `libc.so` wird weiterhin genommen und ist entweder
  - umbenannte `libc_`$< ver >$
  - symlink auf die `libc`, die der Linker nehmen soll

3.3 SONAMEs

Das ist allerdings alles eher unbefriedigend, daher wurde eine andere Lösung gefunden. In jeder `libc` gibt es ein `SONAME`-Tag, das die jeweilige `libc` kennzeichnet. Meistens ist dies die Versionsnummer, also z.B. „2.3.3“.[2] Dieser Wert wird in jedes Programm mit eingefügt. Dadurch kann jetzt zur Laufzeit festgestellt werden welche `libc` benötigt wird. Das kann zu verschiedene Ergebnissen führen:

benötigte `libc`	**vorhandene** `libc`	**Reaktion**
2.3.3	2.3.3	Keine Probleme. Programm startet.
2.3.3	2.2.7	Das Programm startet zwar, aber mit Fehlermeldung.
2.3.3	2.4.9	Keine Probleme. Programm startet.
2.3.3	3.1.3	Das Programm wird nicht starten, da sich evtl. verwendete Schnittstellen geändert haben.

[2] Historisch bedingt heißt die „`glibc 2`“ intern „`libc 6`“. Die `SONAME`s sind also unabhängig vom Dateinamen. Von dieser Möglichkeit sollte aber so wenig wie möglich Gebrauch gemacht werden.

Im Fall der vorhandenen Version 2.4.9 startet das Programm, da nur Fehler behoben oder lib-interne Dinge verändert wurden. Die Schnittstellen nach außen haben sich nicht verändert. Es sollte trotzdem möglichst die selbe Minor-Version verwendet werden, um Probleme mit NLS[3] zu vermeiden.

3.4 Finden der shared objects

Wie aber werden die vorhandenen `libc`-Files überhaupt gefunden? Der shared linker, der bei jedem Programmaufruf aufgerufen wird, sucht in bestimmten Verzeichnissen. Eine Liste aller Verzeichnisse mit shared objects steht in der `/etc/ld.so.conf`. Die Einträge `/lib` und `/usr/lib` werden üblicherweise weggelassen, da diese standardmäßig durchsucht werden. Zusätzlich findet man dort z.B. oft noch den Eintrag `/usr/X11R6/lib`.

3.4.1 Symlinks im Suchpfad

Aber auch das ist noch nicht optimal, da das Suchen sehr zeitaufwändig ist. Daher gibt es zusätzlich noch Symlinks:

`lib_2.3.so`	→	`lib_2.3.3.so`	(`lib_2.3.so` zeig auf höchste `lib_2.3.x.so`)
`lib_2.so`	→	`lib_2.3.3.so`	(`lib_2.so` zeig auf höchste `lib_2.x.x.so`)
`lib_1.5.so`	→	`lib_1.5.17.so`	(`lib_1.5.so` zeig auf höchste `lib_1.5.x.so`)
`lib_1.so`	→	`lib_1.5.17.so`	(`lib_1.so` zeig auf höchste `lib_1.x.x.so`)

3.4.2 Der Cache `/etc/ld.so.cache`

Um nicht bei jeden Prgrammaufruf alle Verzeichnise, die in `/etc/ld.so.conf` angeführt sind, wieder rekursiv durchsuchen zu müssen, legt das Programm `ldconfig` einen Cache an (üblicherweise in der Datei `/etc/ld.so.cache`). Dabei handelt es sich um ein binäres Datenbank-Format, das die Suche erheblich beschleunigt.

3.4.3 Die Umgebungsvariable `LD_LIBRARY_PATH`

Zusätzlich zu den Suchpfaden in `/etc/ld.so.conf` kann ein Benutzer auch ohne root-Rechte mittels der Umgebungsvariable `LD_LIBRARY_PATH` eigene Suchpfade hinzufügen. Diese Pfade werden allerdings nicht gecached. Eine weitere Möglichkeit um den Library Suchpfad anzugeben ist, den gcc Parameter „`-rpath`“ zu benutzen. So kann der Pfad fest einkompiliert werden. Dies ist allerdings meist nur für Eintwickler interessant.

3.5 Funktionsweise

Wie funktionieren nun diese Shared Objects eigentlich? Der Linker kopiert bei shared linked Programmen einen Loader für `.so`'s mit in jedes Programm. Damit wäre allerdings der Vorteil der leichten Wartbarkeit und der Platzersparnis bei solchen Programmen wieder fast aufgehoben, denn nun befindet sich wieder der gleiche Code in allen Programmen. Die Lösung hierfür is, dass nur ein „Micro Shared Object Loader“ an jedes Programm anghängt wird. Dieser ist sehr klein und lädt dann den eigentlichen Loader (meist `/lib/ldlinux.so`) nach. Dieser Loader lässt sich auch als gcc Parameter angeben.

[3] Native Language Support

4 Updaten der `libc`

Was ist zu Beachten, wenn man die `libc.so` updaten will? Natürlich sollen andere Benutzer, die auf dem Rechner eingeloggt sind möglichst nichts davon mitbekommen:

Wenn ein neues **Major Release** installiert werden soll, also z.B. Update von Version 1.x auf Version 2.x, muss man lediglich die neue `libc` in das `/lib`-Verzeichnis dazukopieren. Dann kann mit einer atomaren Operation der Symlink umgesetzt werden.

Bei einem **Minor Release** Update verläuft es ähnlich, hier muss ebenfalls nur ein Symlink umgesetzt werden.

Problematisch ist es allerdings bei einem **Bugfix Release**. Das `ln` Kommando kann das nicht als einzelne, atomare Operation durchführen, es wären also Inkonsistenzen zu erwarten. Aus diesem Grund setzt das vorher besprochene `ldconfig` auch die Symlinks korrekt. Und ein Aufruf von `ldconfig` ist praktisch atomar.

Bei einem **Downgrade** muss allerdings weiter selbst Hand angelegt werden. Dies ist allerdings mehr Aufgabe des Entwicklers, den Benutzer einer Distribution kümmert das in der Regel nicht.

Damit `ldconfig` unabhängig von der installierten `libc` Version funktionieren kann ist es statisch gelinkt. `ldconfig` selbst ist also von einem Update nicht betroffen.