Bibliotheken

Autor: Steffen Dettmer (*steffen@dett.de*)
Layout: Torsten Hemm (*T.Hemm@gmx.de*)

Inhaltsverzeichnis

1 Einleitung

Dieses Kapitel erklärt Sinn, Zweck und Handhabung von Systembibliotheken für Anwender. Softwareentwickler können diesen Artikel als Einführung in die Materie verwenden, werden jedoch keine Details über Bibliotheken aus Programmierer- oder Entwicklersicht finden.

Benutzer und Anwender von Linuxsystemen finden hier einen verständnisfördernden Überblick. Systemadministratoren gewinnen einen Einblick in die Zusammenhänge unter denen für sie interessanten Gesichtspunkten, werden jedoch weitere Lektüre benötigen, um Einzelheiten zu denen sie interessierenden Punkten zu erfahren.

2 Komplexe Systeme

Ein Komplexes System besteht in der Regel aus mehreren, kleineren Systemkomponenten oder Subsystemen. Diese können wiederum aus kleineren Komponenten bestehen. Nur durch das Zusammenspiel vieler kleiner Komponenten lassen sich große Systeme beherrschen. Häufig ergibt es sich, dass einfach kleine Komponenten von mehreren komplexeren Komponenten benutzt werden.

Sieht man sich zum Beispiel verschiedene X-Anwendungen an, so fallen viele Ähnlichkeiten unter den Anwendungen auf. So kann beispielsweise jede Anwendung durch Maus oder Tastatur gesteuert werden, Anwendungen haben Fenster mit Rahmen und Funktionssymbolen und reagieren auf verschiedene Ereignisse (zum Beispiel Mausklicks).

Hier erkennt man, dass selbst grundverschiedene Anwendungen häufig in Details ähnliche Funktionalitäten haben.

2.1 Systemkomponenten

Ein Linux- beziehungsweise Unix-System besteht aus sehr vielen Komponenten. Dabei lässt sich ein solches System auf mehrere Arten in Komponenten zerlegen. Die Art der Zerlegung reflektiert eine bestimmte Sicht auf das System. Eine Zerlegung könnte man aus Sicht der Hardware vornehmen. Ein Linuxsystem besteht nach dieser Sicht aus Monitor, Tastatur und Rechner. Der Rechner besteht wiederum aus Einsteckkarten, Festplatten und einem Netzteil. Eine Einsteckkarte besteht aus einer Leiterplatte und etlichen Schaltkreisen.

Je nach Ebene, auf der man sich bei einer Systemanalyse befindet, interessieren einen mehr oder weniger Details. Von ganz oben gesehen besteht im Beispiel das System aus Rechner und Monitor, die über ein Kabel verbunden sind. Das dies überhaupt nur funktioniert, weil eine Grafikkarte in den Rechner eingebaut ist, die Speicherschaltkreise, Oszillatoren und viele andere Bauteile enthält, und wie nun eigentlich ein Speicherschaltkreis funktioniert, ist auf dieser Ebene unerheblich. Jedoch ist es von elementarer Bedeutung, dass eine Grafikkarte Speicherschaltkreise hat, wenn man sich auf einer Ebene befindet, die den Aufbau von Grafikkarten zeigt.

Das Interessante an den Ebenen des Systems ist, das jede Systemkomponente auch als System angesehen werden kann (daher auch der Name Subsytem). Für Grafikkartenhersteller ist eine VGA-Karte ein vollständiges System. Für einen Linuxanwender ist ein Computersystem (Rechner, Monitor usw.) ein vollständiges System. Diese Struktur lässt sich jedoch auch nach oben hin weiterführen: Für einen Mitarbeiter einer IT Abteilung ist erst ein Netzwerk mit vielen daran angeschlossenen Computersystemen ein vollständiges System; einzelne Server (also Computersysteme) sind für ihn eine Komponente. Auch hier gilt der Komponentensichtbereich: Sicherlich ist einem IT Mitarbeiter klar, dass ein Server üblicherweise über Speicherschaltkreise verfügt, aber es interessiert ihn überhaupt nicht.

Dieses Strukturierungsbeispiel reflektiert die Hardware-Sicht. Neben vielen weiteren kann man eine solche Strukturierung auch aus Software-Sicht vornehmen. Hier spricht man von Anwendungen als Systeme, die aus Unteranwendungen oder Programmen bestehen, diese wiederum bestehen vielleicht aus Funktionen. Anwendungen können auch andere Anwendungen verwenden.

Hier findet man eine weitere interessante Eigenschaft: die einzelnen Sichten untereinander interessieren sich in der

Regel nicht für andere Sichten. Betrachtet man eine Anwendung, so ist sicherlich klar, das diese irgendwo wieder in Speicherschaltkreisen lebt, aber das interessiert in dieser Sichtweise überhaupt nicht. Das gibt die Möglichkeit, sich auf die essentiellen Aspekte von Systemebenen zu konzentrieren. Dadurch kann man auch komplexe Systeme verstehen. Eine typische X-Anwendung verwendet möglicherweise Millionen von Maschineninstruktionen, die in Millionen von Speicherstellen von Speicherschaltkreisen gespeichert sind. Das ist viel zu komplex, um von Menschen überblickt werden zu können. Aber jede Ebene für sich betrachtet, kann verstanden und beherrscht werden.

Auch bei der Softwaresicht lässt sich die Strukturierung nach oben hin erweitern: Man spricht beispielsweise von verteilten Systemen, deren Komponenten verschiedene Anwendungen (zum Beispiel Datenbanken und Webserver) sind, die auf verschiedenen Servern laufen (wobei das mit den Servern jedoch schon wieder uninteressant ist).

Im folgenden wird die Softwaresicht etwas detaillierter beschrieben.

2.1.1 Systemkomponenten von oben gesehen

Betrachtet man eine Anwendung, kann man Komponenten finden, aus denen die Anwendung besteht. Ein *KDE* Programm beispielsweise besteht aus Fenstern, die Menüs besitzen können und einen Rahmen haben. Ein Menü besteht aus Einträgen und Ereignisverarbeitungsfunktionen, die dafür sorgen, dass auch etwas passiert, wenn man eine Menüfunktion auswählt. Ein Rahmen besteht nun aus einer Titelleiste, einigen Knöpfen und vielleicht einer Laufleiste.

Betrachtet man das System genauer, stellt man schnell fest, dass etliche dieser Komponenten bei allen *KDE* Anwendungen sehr ähnlich sind. So hat fast jedes *KDE* Programm ein Fenster mit einem Titel, und wenn zwei Fenster Laufleisten haben, sehen diese sehr ähnlich aus. Diese Eigenschaften sind also bei allen *KDE* Programmen ähnlich.

Geht man eine weitere Ebene tiefer, stellt man fest, dass man viele Eigenschaften findet, die bei den eben beschriebenen Komponenten ähnlich sind. Zum Beispiel besteht eine Titelleiste und eine Laufleiste aus Bildpunkten, die über Grafikkarten dargestellt werden. Das hört sich zunächst banal an, aber wenn man bedenkt, dass diese Grafikkarten in verschiedenen Systemen eingebaut sein können und über Netzwerke hinweg dargestellt werden können, kommt man zu der Erkenntnis, dass es gar nicht so banal ist, einen Bildpunkt in die richtige Grafikkarte zu schreiben. Auf Ebene der Bildpunkte ist dies jedoch eine interessante Frage: Es muss etwas geben, was Informationen und Netzwerkverbindungen verwaltet. Dabei spielen Zugriffsbeschränkungen und viele andere Fragen eine Rolle.

Neben grafischen Elementen gibt es noch weitere ähnliche Eigenschaften. Viele *KDE* Programme können zum Beispiel verschiedene Arten von Konfigurationsdateien verwenden. Diese Dateien müssen irgendwie geöffnet und eingelesen werden. Die Daten müssen dann weiterverarbeitet werden, es müssen Fehler erkannt werden, zum Beispiel wenn das Dateiformat ungültig ist. Hier gibt es Komponenten, die die Konfigurationsoptionen aus Dateien einlesen und verarbeiten.

Diese Komponenten verwenden Dateioperationen. Auch viele andere Komponenten verwenden Dateioperationen, darunter selbstverständlich auch viele Anwendungen oder Komponenten, die überhaupt nichts mit dem *KDE* zu tun haben.

Eine Dateioperation wiederum verwendet Systemfunktionen, um auf die Datenbytes einer Datei nach dem Öffnen zugreifen zu können. Diese Systemfunktionen verwenden Systemtreiber, um die Daten beispielsweise von Festplatten lesen zu können. Die Systemtreiber kann man intern auch als System betrachten. Diese Sicht wird vermutlich von einem Festplattentreiberprogrammierer verwendet werden. Für ihn verwendet sein System (der Treiber) viele Funktionen, darunter auch andere Systemfunktionen.

Die Treiber, die Teil des Systemkerns (kernels) sind, verwenden Maschineninstruktionen und Maschinenkommunikationsmittel. Dies ist die unterste Softwareebene, die es bei Personalcomputern (PCs) gibt. Der zentrale Schaltkreis, das Herz eines PCs, die CPU (Central Processing Unit) kann ganz einfache Maschinenbefehle ausführen und über Steuerleitungen mit anderen Geräten kommunizieren.

2.1.2 Systemkomponenten von unten gesehen

Nun stellt sich möglicherweise dennoch die Frage: wenn eine CPU mit einer Festplatte kommunizieren kann, warum dann diese ganzen Systemebenen? Warum verwendet man dann Treiber, wenn eine Anwendung doch auch direkt mit der Festplatte kommunizieren kann?

Mindestens eine Antwort sollte nach kurzer Überlegung gefunden werden können. Natürlich würde die Anwendung viel zu komplex. Anstatt einer Komponente zu sagen "lade mir mal den Parameter x aus der Datei y" müsste sie Hunderte von Maschineninstruktionen verwenden, um allein die Datei zu finden. Weiterhin müsste sie alle Geräte direkt unterstützen; kommt eine neue Festplatte auf den Markt, müssten unter Umständen sämtliche Anwendungen angepasst werden!

Die Klasse der Festplattentreiber haben eines gemeinsam: im Wesentlichen können sie Datenpakete an bestimmte Stellen schreiben oder von diesen Lesen. Diese Stellen haben nichts mit Unterverzeichnissen oder Dateien zu tun: es sind einfach nur Nummern. Ein Datenpaket hat auch eine feste Größe. Auf dieser Ebene ist es sehr umständlich, Dateien zu verwalten: Man muss sich um Listen mit den Speicherstellennummern kümmern, alle diese einlesen und so weiter. Wie man eine solche Speicherstelle liest, ist zu dem noch für verschiedene Geräte (wie IDE oder SCSI Festplatten) unterschiedlich. Daher fasst man diese Funktionalität in Treibern zusammen.

Hat man solche Treiber, die sich um die gesamte Gerätekommunikation kümmern, kann man darauf aufsetzen und ein Dateisystem bauen. Das sorgt dafür, dass man sich nicht mehr um Tausende Details kümmern muss, sondern einfach beliebig lange Daten (anstatt Datenpakete fester, jedoch geräteabhängiger Größe) über Namen (anstatt Nummern) lesen und schreiben kann.

Man kann nun auch Dateisysteme bauen, die nicht direkt auf Gerätetreiber zugreifen, sondern die Lese- und Schreibanfragen über Netzwerkgerätetreiber an andere Server weiterreichen und hier über Gerätetreiber ausgeführt werden. Damit kann man auch auf Dateien zugreifen, die im Netzwerk liegen.

Diese ganzen Funktionen spielen sich im Kernel ab. Anwendungsprogrammierer interessieren sich nicht dafür, wie ein Kernel eine Datei öffnet. Sie interessieren sich nur dafür, wie sie diese Kernelfunktion aufrufen können. Hierzu gibt es dann Kernelfunktionen, um zum Beispiel eine Datei mit einem bestimmten Namen zu öffnen und Daten aus ihr zu lesen.

Diese Programmierschnittstelle ist jedoch immer noch unkomfortabel. Zum Beispiel ist es sehr langsam, eine solche Funktion aufzurufen. Möchte man 100 Zeichen einzeln aus einer Datei lesen, so muss man sinnvollerweise sofort 100 Zeichen lesen, irgendwo zwischenspeichern und dann erst verarbeiten. Die Kernelschnittstelle umfasst auch nur relativ wenige, ziemlich "dumme" Funktionen.

Da alle Programmierer diese Probleme haben, gibt es fertige Funktionen, die diese lösen. Diese Funktionen sind bequemer zu verwenden und bieten weitere Vorteile.

Nun könnte natürlich jedes Programm diese Funktionen als Kopie mitbringen. Die Nachteile hierfür sind wieder klar: bei Änderungen müsste man alle Programme anpassen, die Programme würden sehr groß und kompliziert werden und so weiter.

Also fasst man diese grundlegenden Funktionen mit vielen anderen, die indirekten, aber komfortableren Zugriff auf Kernelfunktionen bieten, zusammen. Solche Funktions-Zusammenfassungen nennt man Bibliotheken. Die Zusammenfassungen von den hier genannten Basisfunktionen nennt man C-Bibliothek. *C* ist eine hardwarenahe Programmiersprache, in der der Unix-Kernel und die C-Bibliothek geschrieben sind. Dieser Name deutet an, dass hier eine C-Programmierschnittstelle bereitgestellt wird, also können in C geschriebene Programme einfach darauf zugreifen. Da C-Programme ja auch "nur" in Maschineninstruktionen übersetzt werden (eine CPU kann ja nur diese Maschineninstruktionen ausführen), kann auch jede andere Sprache, die in solche Maschineninstruktionen übersetzt wird, hierauf zugreifen. In der Praxis sind das demzufolge eigentlich alle Sprachen (da eine CPU ja nur eine Maschinensprache kennt!). "Eigentlich" zeigt, dass es auch Ausnahmen gibt, wie zum Beispiel *Java*. *Java* läuft auf einer sogenannten virtuellen Maschine, die eine eigene Maschinensprache hat. Natürlich werden diese indirekt auch wieder "nur" auf Maschineninstruktionen abgebildet, doch eine genauere Betrachtung würde den Rahmen dieses Dokumentes sprengen.

Diese C-Bibliothek ist so grundsätzlich, dass in der Praxis so gut wie jedes Programm direkt oder indirekt auf sie zugreifen.

In unserem Beispiel sahen wir aber, dass eine Anwendung auf noch speziellere und komfortablere Funktionen zurückgreift. Wenn man beispielsweise Konfigurationsdateien verarbeiten möchte, kann man natürlich in jede Anwendung die vielen hundert erforderlichen Schritte einprogrammieren. Aber es gibt ja eine bessere Lösung: Man baut eine Bibliothek dafür! Diese Bibliothek liest intern einzelne Zeichen aus Dateien und verarbeitet diese. Verwendet man Funktionen dieser Bibliothek, muss man sich gar nicht mehr darum kümmern, dass man mehrere Funktionen benötigt, um die Daten zu lesen. Man ruft einfach eine Funktion auf, die das weitere erledigt.

3 Motivation für Bibliotheken

In den vorherigen Absätzen wurden schon etliche Gründe genannt, warum Bibliotheken sinnvoll sind. Man kann Funktionen so bereitstellen, dass andere Anwendungen oder Komponenten diese verwenden. Änderungen oder Erweiterungen müssen nur an einer zentralen Stelle (in der Bibliothek) vorgenommen werden. Die Wahrscheinlichkeit von Programmfehlern ist geringer, da man durch die häufige Verwendung der selben Funktionen Fehler schneller finden kann. Das Entwickeln von Anwendungen ist einfacher und schneller, wenn man auf komfortable Schnittstellen von Bibliotheksfunktionen zurückgreifen kann. In der Praxis geht man noch einen Schritt weiter: es gibt Bibliotheken mit identischen Schnittstellen für verschiedenste Systeme. Beispielsweise unterschieden sich die Kernel von Unix- und Windowssystemen erheblich, jedoch sind die gleichen C-Bibliotheksfunktionen vorhanden. Das C-Programm, das nur C-Bibliotheksfunktionen verwendet, kann auf Windows und Linux übersetzen werden, ohne das man es anpassen muss, selbst wenn indirekt unterschiedliche Kernelfunktionen von der C-Bibliothek verwendet werden. Man benötigt eben nur eine passende Bibliothek.

Die Verwendung von Bibliotheken kann weitere Vorteile haben. So spart man beispielsweise Speicherplatz, wenn man die Funktionen nur einmal hat, anstatt in jeder Programmdatei. Auch bei Aktualisierungen wie zum Beispiel Sicherheitslücken ist das Hilfreich: man aktualisiert die betroffene Bibliothek, und alle Programme, die diese verwenden, verwenden automatisch die neue, sicherere Funktion.

4 Arten von Bibliotheken

Es gibt sehr viele Arten von Bibliotheken. Diese lassen sich unter verschiedenen Gesichtspunkten einteilen. Man kann sie nach Funktionsart einteilen. So gibt es zum Beispiel Bibliotheken für grafische und mathematische Funktionen. Man kann sie auch nach der Programmiersprache einteilen. Bisher bezogen sich alle Beispiele auf Systembibliotheken, die aus *C* heraus sehr bequem, aber auch aus anderen Sprachen heraus aufgerufen werden können. Es gibt aber auch Sprachbibliotheken, die nur für bestimmte Sprachen zur Verfügung stehen. So gibt es beispielsweise sehr viele Bibliotheken für *Perl* oder *Java*, die man aus anderen Sprachen heraus kaum verwenden kann.

Dieser Artikel beschäftigt sich mit Systembibliotheken. Auch diese kann man in verschiedene Klassen einteilen, ein Beispiel gab es bereits. Eine weitere Einteilung ist etwas schwieriger zu verstehen: man unterscheidet Bibliotheken danach, wann sie zu einem Programm hinzugebunden werden. Hier gibt es drei Zeitpunktklassen: direkt nach der Programmübersetzung, bei jedem Programmstart oder zu einem beliebigen Zeitpunkt während der Programmlaufzeit (also den Zeitpunkten, an denen das Programm läuft). Analog unterscheidet man drei Bindungsarten, die im folgenden kurz genannt werden. Die Namen dieser Bindungsarten mögen etwas verwirrend erscheinen, hiervon sollte man sich nicht beeindrucken lassen.

4.1 Statische Bindung

Die erste Möglichkeit wurde bisher etwas unterschlagen: Das Binden von Programmen und Bibliotheken zur Entwicklungszeit. In diesem Fall wird das Programm und die Bibliothek fest verbunden. Anschließend wird die Bibliotheksdatei nicht mehr für das Programm benötigt, da die verwendeten Funktionen in die Datei kopiert werden. Man spricht hier vom statischen Linken (link eng: verbinden).

Programme, die statisch mit Bibliotheken verbunden sind, nutzen einige der Vorteile nicht, so sind sie zum Beispiel größer und die Bibliotheken können vom Anwender nicht so ohne weiteres aktualisiert werden. Deshalb verwendet man dieses Verfahren meist nur für sehr spezielle Programmbibliotheken, die nur von diesem Programm verwendet werden, oder die man aus verschiedenen anderen Gründen fest mit dem Programm verbinden möchte.

Diese Bibliotheken sind technisch gesehen einfache Archive, die viele Objekt-Dateien enthalten können. So ein Archiv ist in etwa mit einem *tar* oder *jar* Archiv vergleichbar (aber üblicherweise unkomprimiert). Die Bezeichnung Archiv sollte hier aber nicht verwendet werden, sie ist irreführend, da man heutzutage sofort an allgemeine Archive wie *tar* oder Zip-Archive denkt.

4.2 Gemeinsame Nutzung

Eleganter ist die Möglichkeit, Programme bei deren Start mit der passenden Bibliothek zu verbinden. Dies nennt man dynamisches Binden. Dies ist ein komplizierterer Vorgang. Beim Start führt ein besonderes Programm die Verbindung unmittelbar vor dem eigentlichen Programmstart aus. Dazu muss die Bibliothek geladen werden, Einsprungadressen geprüft und weitere Aktionen durchgeführt werden. Dieses Vorgehen ermöglicht es, dass mehrere verschiedene Anwendungen eine Bibliothek gemeinsam nutzen. Die Bibliotheken, die so verwendet werden, nennt man "gemeinsam nutzbare Bibliotheken" (eng: shared libraries). Auch wenn es sich technisch gesehen um dynamisch verbundene Bibliotheken handelt, und sie auch manchmal DLLs (dynamically linked libraries) genannt werden, nennt man sie üblicherweise NICHT dynamische Bibliothek, da dieser Begriff für das folgende Vorgehen reserviert ist.

4.3 Dynamische Bindung

Richtig dynamisch wird es, wenn ein Programm erst zur Laufzeit Bibliotheken lädt und verwendet. Dies ermöglicht eine sehr hohe Flexibilität. Zum Beispiel kann ein Programm unter Umständen teilweise funktionieren, wenn nicht alle benötigten Bibliotheken installiert sind, oder es kann Bibliotheken erst dann laden, wenn sie tatsächlich benötigt werden. Hier spricht man von dynamischen Bibliotheken, oder von dynamisch ladbaren Bibliotheken (eng: dynamically loaded libraries)

Unter Linux (und anderen Systemen) sind die *shared libraries* bzw. *dynamically linked libraries* und *dynamically loaded libraries* die gleichen Dateien und sich sehr ähnlich. Nur die Art der Verwendung unterscheidet sich.

Daher kommt vermutlich auch die unklare Namensabgrenzung: Meint man mit "Bibliotheken" bestimmte Dateien für eine dieser beiden Bindungsarten, so sind das eben dynamische Bibliotheken.

Interessant ist hier ein weiteres Detail. Da jedes Programm, welches dynamisch Bibliotheken laden möchte, sehr ähnliche Funktionen verwenden muss, gibt es auch hierfür eine Bibliothek: eine Bibliothek zum Laden von Bibliotheken. Diese heißt *libdl* (dl engl: *dynamic loading*). Diese Bibliothek wird dynamisch gelinkt und ermöglicht komfortables dynamisches laden (diese Bibliothek dynamisch zu laden, macht verständlicherweise kaum Sinn).

4.4 Beispiele

Die bereits erwähnte C-Systembibliothek, die kurz als *libc* oder *C-Lib* bezeichnet wird, ist fast immer eine dynamische Bibliothek, die dynamisch gelinkt (und nicht geladen!) wird. In der Praxis werden die meisten C-Bibliotheken dynamisch gelinkt, und nur in sehr wenigen, speziellen Fällen macht man vom dynamischen Laden Gebrauch.

Die Bibliotheken, die Zugriff auf X-Window-System-Funktionen gestatten (*X-Lib* genannt), fallen eben so in diese Kategorie wie die Mathematikbibliothek *math*, die Funktionen wie Sinus und Cosinus bereitstellen.

5 Gemeinsam genutzte, dynamisch gelinkte Bibliotheken

5.1 Verwenden von Bibliotheken

Hat man eine Linux-Distribution (*SuSE Linux*, *RedHat Linux* oder andere) installiert, so hat man sehr viele Komponenten installiert. Neben dem Kernel (also Linux selbst) sind unzählige Programme und natürlich Bibliotheken installiert worden. Die Bibliotheksdateien befinden sich in den Verzeichnisse **/lib** und **/usr/lib**. Etliche der installierten Programme verwenden auch statisch gebundene Bibliotheken, aber da man davon im Betrieb nichts merkt, wird das hier nicht näher betrachtet.

Die dynamischen, gemeinsam benutzbaren Bibliotheken sind für alle Programme des Linuxsystems verwendbar. Neben der einfachen Benutzung bietet ein Linuxsystem jedoch weitere Vorteile, die anspruchsvolleren Anforderungen genügen.

Es ist möglich, zu einem Zeitpunkt verschiedene Versionen einer Bibliothek installiert zu haben. Man kann in diesem Fall beeinflussen, welche Bibliotheken ein Programm verwendet. So kann man auch ältere Programme weiterverwenden, die eine alte, inkompatible Version einer Bibliothek benötigen.

Man kann auch veranlassen, dass bestimmte Bibliotheken, die ein Programm verwendet, gegen andere ersetzt werden, um damit beispielsweise gezielt einzelne Funktionen zu ersetzen.

5.2 Der Linux Programm Lader

Soll ein Programm, welches Bibliotheken verwendet, gestartet werden, so müssen natürlich diese mitgeladen werden. Dazu wird zunächst gar nicht das eigentliche Programm gestartet, sondern der sogenannte Programm *Lader* (eng: *program loader*). Dieser wird auch *dynamic linker* genannt, da er die Bibliotheken zum Programm bindet (engl: link). Auf Linuxsystemen heißt dieser **/lib/ld-linux.so.2** (die .2 ist wieder eine Versionsnummer; das "-linux" zeigt, dass er Linuxspezifisch ist. Oft wird er jedoch insbesondere in technischer Dokumentation *ld.so* genannt). Der Programmlader lädt nun auch die benötigten Bibliotheken automatisch. Im Kopf der Programmdateien steht eine Liste mit diesen. Nach dem Laden bereitet er das Programm zum Start vor, und startet es letztendlich.

Dieser Vorgang findet bei fast allen Linux-Programmen statt. Nur sehr wenige Programme verwenden keine dynamischen Bibliotheken. Beispiele dafür sind `login` (hier aus Sicherheitsgründen) und `rpm` (damit man es auch starten kann, wenn die Systembibliotheken beschädigt oder defekt sind).

Um eine Bibliothek zu laden, bestimmt der Programm Lader den Namen der Bibliotheksdatei.

5.3 Namen von Bibliotheken

Jede Bibliothek hat natürlich einen Namen. So heißt die Lib-C beispielsweise *c* und Mathematikfunktionen findet man als *m* und die für das dynamische Laden heißt *dl* (C-Programmierer scheinen sehr kurze Namen zu lieben).

Vor diese Namen wird **lib** vorangestellt, um kenntlich zu machen, dass es sich um eine Bibliothek handelt. Dahinter schreibt man bei gemeinsam benutzbaren Bibliotheken **.so** (für engl: *shared object*) und bei statisch linkbaren **.a** (für engl. *archive*).

Hinter das **.so** hängt man noch eine Versionsnummer an. Diese Versionsnummer wird durch einen Punkt abgetrennt und kann mehrwertig sein, so kann man beispielsweise eine **/lib/libdl.so.1.9.9** haben. Hierbei handelt es sich um die *dl* in der Version 1.9.9.

Da die Bibliotheken die Versionsnummer im Dateinamen haben, aber ein Programm in der Regel nicht an einer bestimmten Version interessiert ist, werden symbolische Links angelegt (wie diese Links entstehen, wird später noch beschrieben). So wird beispielsweise die Datei **libcrack.so.2.7** auch als **libcrack.so.2** und als **libcrack.so** bekannt

gemacht. Ein Programm kann nun gegen *libcrack* gelinkt werden, und es wird auch noch funktionieren, wenn eine **libcrack.so.2.8** installiert wird, oder ein System nur eine **libcrack.so.2.6** anbietet. Linkt ein Programm direkt gegen eine Version, so muss natürlich die genau passende vorhanden sein. Dies wird auch manchmal gemacht, was bei Updates zu Problemen führen kann, weil die Programme plötzlich nicht mehr gestartet werden können. Oft funktionieren die Programme jedoch auch mit neueren Versionen. Nun kann man natürlich einfach einen Link per Hand erzeugen, der von Programm verwendet wird, und auf eine vorhandene, neuere Version zeigt, doch kann es auch hier zu Problemen kommen. Diese Vorgehen sollte daher in Produktion vermieden werden.

5.4 Platzierung im Dateisystem

Werden diese bereits für das Starten eines Linuxsystems benötigt, so werden diese Bibliotheken in **/lib** installiert. Andere, die zum Starten selbst noch nicht benötigt werden (zum Beispiel Grafikbibliotheken) finden sich dagegen in **/usr/lib**. Zusätzliche Bibliotheken, die nicht zu einer Distribution gehören, sondern selbst installiert wurden, finden sich meistens in **/usr/local/lib**.

5.5 Funktion des Dynamischen Linkers (Programm Laders)

Der Programmlader versucht zunächst, sämtliche Bibliotheken zu laden. Er sucht dazu in den Pfaden, die in der Umgebungsvariablen **LD_LIBRARY_PATH** hinterlegt sind. Anschließend wird in einem Cachefile gesucht (**/etc/ld.so.cache**), welches aus Effizienzgründen verwendet wird, und letztlich in **/usr/lib** und **/lib**. Da die Verwendung von **LD_LIBRARY_PATH** im Normalbetrieb umständlich ist, wird diese normalerweise nicht gesetzt, vielmehr stehen alle Bibliotheken im Cachefile. Dieses wird über ein spezielles Werkzeug erstellt. Fast alle Bibliotheken werden über diesen Weg geladen.

Findet der Dynamische Linker nicht alle benötigen Bibliotheken (oder fehlt eine Funktion oder ein anderes Symbol in einer gefundenen Bibliothek), so wird der Start mit einer Fehlermeldung abgebrochen.

5.6 Umgebungsvariablen

Der Dynamische Linker wird über Umgebungsvariablen gesteuert. Neben der bereits erwähnten Variable **LD_LIBRARY_PATH** erkennt der Programm Lader die Variable **LD_PRELOAD**. Die durch Leerzeichen getrennten Einträge dieser Variable ist eine Liste von Bibliotheken, die auf jeden Fall zuerst geladen werden sollen. Dies ermöglicht es, Bibliotheken zu laden, die Funktionen von anderen Bibliotheken "überschreiben". Der Linker verbindet die Programmfunktionen in diesem Fall nämlich mit denen aus dieser Bibliothek, selbst wenn später eine andere Bibliothek diese Funktion auch definiert. Dies ist jedoch auch für Angriffe ausnutzbar, so könnte man beispielsweise Bibliotheken "unterschieben", die über Seiteneffekte von "normalen" Funktionen bestimmte ungewollte Aktionen ausführen. Aus diesem Grund ignoriert der Programmlader (beide) Variablen bei "setuid" Programmen (also Programme, die unter anderen Benutzerrechten laufen, als der Aufrufer besitzt). Dadurch kann ein Aufrufer sich keine zusätzlichen Rechte verschaffen.

Eine Liste aller Umgebungsvariablen findet man in der Manpage **ld.so**.

5.7 Cachefile erzeugen

Das bereits erwähnte Cachefile wird vom Werkzeug `ldconfig` erzeugt. Dieses verwendet eine Konfigurationsdatei **/etc/ld.so.conf**. Erkennt man, dass `ldconfig` und **ld-linux.so** sehr stark zusammenarbeiten, wird auch der Name der Datei erkenntlich. Indirekt konfiguriert man ja den dynamischen Linker (der ja manchmal **ld.so** genannt wird).

`ldconfig` sucht in den Verzeichnissen, die in **/etc/ld.so.conf** stehen, nach Bibliotheken. Bei diesem Vorgang werden auch die symbolischen Links angelegt, die im Abschnitt "Namen von Bibliotheken" beschrieben wurden. Die gefundenen Bibliotheken aus allen angegebenen Verzeichnissen werden mit Versions- und weiteren Informationen im Cachefile **/etc/ld.so.cache** abgespeichert (welches dann vom Programmlader verwendet wird).

Das Programm `ldconfig` lässt sich über Kommandozeilenparameter steuern, die in der Manpage gelistet sind. Ein

Beispiel ist der Schalter `-v`, der etliche Informationen anzeigt; ein Aufruf von `ldconfig -v` ist vielleicht an dieser Stelle interessant.

Aus naheliegenden Gründen verwendet `ldconfig` selbst keine dynamischen Bibliotheken, sondern ist statisch gebunden.

6 Verwalten von Bibliotheken

Ein Administrator hat mit Bibliotheken üblicherweise recht wenig Arbeit. Wird ein neues Verzeichnis mit Bibliotheken eingerichtet, beispielsweise **/usr/local/lib**, so muss es in die Datei **/etc/ld.so.conf** eingetragen werden. Anschließend ist natürlich das starten von `ldconfig` erforderlich.

6.1 Installieren von Bibliotheken

Meistens installiert man Bibliotheken über RPMs oder andere Pakete. Diese Pakete führen manchmal `ldconfig` automatisch aus, jedoch sollte man nach dem Installieren sicherheitshalber immer `ldconfig` ausführen.

6.2 Aktualisieren von Bibliotheken

Das Aktualisieren von Bibliotheken ist zunächst sehr einfach, wenn man RPM oder andere Paketverwaltungswerkzeuge benutzt. Diese überschreiben in der Regel die alten Versionen automatisch. Manchmal wird `ldconfig` automatisch gestartet, aber man sollte es auch hier sicherheitshalber immer per Hand nach dem Aktualisieren ausführen.

Hier können sich aber einige Schwierigkeiten ergeben, insbesondere, wenn man häufig benutzte Bibliotheken aktualisiert. Meistens klappt dies einfach, treten jedoch Probleme auf, kann eine Reparatur aufwendig werden. Am kritischsten sind Erneuerungen der *libc*, da diese von praktisch allen Kommandos verwendet wird. Ohne diese Bibliothek funktionieren so elementare Kommandos wie `ls` oder `cp` unter Umständen nicht mehr (`ls` ist jedoch meistens ein Shellkommando; `/bin/ls` wird zum Beispiel von der Bash nicht verwendet, weil Bash ein "eigenes" `ls` hat).

Wenn man wirklich wichtige Systeme aktualisiert, empfiehlt es sich, hier einige Vorsichtsmaßnahmen zu ergreifen, die jedoch schon recht tiefes Systemverständnis erfordern. Zunächst sollte der gewissenhafte Administrator die Stand-Alone-Shell *sash* installiert haben. Diese wird in der Regel statisch gelinkt, dass heißt, sie funktioniert komplett ohne Bibliotheken. Sash hat den großen Vorteil, sehr viele Kommandos (in beschränkter Form) fest eingebaut zu haben. Vor einem Update sollte man die *sash* in einem oder zwei erst mal unbenutzten Fenstern öffnen. Dann macht man Sicherheitskopien der Dateien, die das Update verändert. Neben den eigentlichen Bibliotheken sollte man **/etc/ld.so.conf** und **/etc/ld.so.cache** sichern. Geht beim folgenden Update etwas schief, kann man unter Umständen keine Programme mehr starten, die die aktualisierten Bibliotheken verwenden. Dies erkennt man z.B. daran, dass `cp` nicht mehr funktioniert.

In diesem Fall darf man nun keinesfalls "sicherheitshalber erst mal neu starten", wie man es möglicherweise von Windows gewöhnt ist! Das System würde nämlich auch nach einem Start nicht funktionieren; es würde überhaupt nicht hochfahren!

Zunächst sollte man (nochmals) `ldconfig -v"` ausführen (das funktioniert ja immer). Bringt das keine Abhilfe, hat man ja glücklicherweise ein oder zwei *sash* zur Hand. Da diese beispielsweise `cp` eingebaut hat, kann man die Sicherheitskopien wieder aktivieren, obwohl `/bin/cp` nicht mehr gestartet werden kann. Da die *sash* möglicherweise nicht ganz intuitiv bedienbar ist, sollte man vorher etwas mit ihr arbeiten.

Hinweise (falls man vorher nicht mir ihr gearbeitet hat, und man *sash* gerade nicht funktioniert): *Sash* kennt ein `help` Kommando. Muss man eingebaute Kommandos verwenden, zunächst `aliasall` eingeben. Erklärungen hierzu finden sich in help und in der man page (die nach dem Zurückkopieren der Sicherheitskopien wieder funktionieren wird).

Nach dem Wiederherstellen der lib*so* Dateien aus den Kopien, sollte ein Start von `ldconfig` wieder für ein funktionierendes System sorgen. Das kann man z.B. durch das Ausprobieren von Kommandos prüfen ("`/bin/echo hallo`" usw.).

Hat man die Bibliotheken erfolgreich aktualisiert, und geprüft, ob das System funktioniert, sollte man bei nächstmöglicher Gelegenheit das System neustarten. Dies ist notwendig, weil Bibliotheken ja beim Programmstart

geladen werden, und damit bereits gestartete Programme noch die alten Versionen verwenden. Ein Neustart ist eine schnelle Methode, alle Programme neuzustarten. Erfahrene Administratoren wissen, wie sie alle Programme einzeln auch ohne System-Neustart neu starten, was auf wirklich wichtigen Systemen die Ausfallzeiten minimiert.

7 Werkzeuge

Es folgen nur Kurzbeschreibungen einiger Werkzeuge, die speziell auf die Arbeit mit Bibliotheken zugeschnitten sind.

7.1 ld-linux.so

Der Programmlader oder Dynamische Linker wurde oben bereits beschrieben. Er ist dafür zuständig, die von einem Programm benötigten Bibliotheken zu laden und zu binden.

7.2 ldconfig

Dieses ebenfalls bereits oben beschriebene Werkzeug erstellt das Cachefile für den Programmlader. `ldconfig` ist statisch gebunden.

7.3 ldd

Dieses Werkzeug zeigt an, ob und welche gemeinsam benutzbaren Bibliotheken ein Programm verwendet. Es gibt die Abhängigkeiten dieser aus. ldd ist statisch gebunden.

Mit `ldd` kann man sich anzeigen lassen, welche Bibliotheken in welchen Versionen ein Programm verwendet. Dabei verwenden fast alle Programme mindestens den Programmlader **/lib/ld-linux.so.2** und die Lib-C **/lib/libc.so.6**. Komplexe Programme verwenden oft 10 und mehr Bibliotheken (z.B. xterm). Mit `ldd` kann man erkennen, ob ein Programm direkt gegen Versionen gelinkt ist, oder ob es überhaupt dynamisch gebunden wird. Das Kommando `ldd /lib/ld-linux.so.2`" sagt beispielsweise *statically linked* (statisch gelinkt) und `ldd /sbin/ldconfig` *not a dynamic executable*. `ldd` zeigt auch, welche Bibliotheken von Bibliotheken verwendet werden, so ist zum Beispiel die Mathematikbibliothek **/lib/libm.so.6** gegen **libc.so.6** gelinkt.

7.4 nm

`nm` zeigt die Symbole, also Funktionen und Variablen, an, die eine Bibliothek nach außen hin verwendet. Das sind zum einem die Symbole, die eine Bibliothek bereitstellt (exportierte Funktionen) und zum anderen die, die sie verwendet (importiert). `nm -u` zeigt die Symbole an, die eine Bibliothek benötigt, und `nm -g` die, die exportiert werden. Dies kann man sich gut am Beispiel libm.so anschauen. `nm -u /lib/libm.so.6` zeigt beispielsweise, dass libm die Standardfehlerausgabe "stderr" aus der libc verwendet und `nm -g /lib/libm.so.6` zeigt, das eine Funktion "tan" bereitgestellt wird (die den Tangens berechnet).

8 Literatur

Details zu den Programmen finden sich in den entsprechenden Manpages. Die Manpage zu *ld-linux.so* heißt unter Umständen kurz *ld.so*.

David A. Wheeler schrieb das *Program Library HOWTO*, das von *Christoph Schönfeld* ins Deutsche übersetzt wurde. Dieses beschreibt Bibliotheken aus Entwicklersicht und zeigt, wie man Bibliotheken erstellt (siehe auch http://www.dwheeler.com/program-library/)