# Dynamische Bibliotheken unter Windows und Linux

Klaus Wachtler

10. November 2002

# Inhaltsverzeichnis

# 1 Allgemeines

Dies ist eine kurze Übersicht über die Verwendung von dynamischen Bibliotheken aus Programmierersicht, und zwar insbesondere für Unix-Systeme (konkret: Linux) und Systeme der CEMENT-Gruppe (Windows CE, ME, NT u.ä.), siehe **A Windows CEMENT** auf Seite 33.

Gedacht ist dieser Text als Kurzanleitung anhand einfacher Beispiele.

Ein kurzer, aber recht informativer Artikel zu der in Linux verwendeten Laufzeitbibliothek *glibc* ist in [Systemzentrale] zu finden, hier stehen auch weitere Informationen zu dynamischen Bibliotheken.

Wesentlich umfangreichere Information findet sich in [CLib.Ref.].

Der vorliegende Text ist
© 2001 Klaus Wachtler
Breidingstr. 17
29614 Soltau
Tel. 05191/70271, 0171/4553039
AnyWare@Wachtler.de

Der Text ist Teil der Vorlesung *Systemprogrammierung* an der Fachhochschule Hannover und zur nichtkommerziellen Benutzung frei. In Zweifelsfällen bitte fragen!

Der Text ist sowohl in gedruckter Form bei AnyWare/Klaus Wachtler erhältlich, als auch unter `http://www.wachtler.de/dynamischeBibliotheken.ps.gz`. zu kopieren (auf einem vernünfigen System also am einfachsten mit dem Kommando `wget http://www.wachtler.de/dynamischeBibliotheken.ps.gz`).

Die HTML-Seiten der Onlinehilfe kann man sich über das Internet betrachten (`http://www.wachtler.de/dynamischeBibliotheken/index.html`), oder die Datei `http://www.wachtler.de/dynamischeBibliotheken\_html.zip` kopieren, mit `unzip dynamischeBibliotheken_html.zip` entpacken und lokal betrachten.

**Voraussetzungen** für ein Verständnis dieses Textes sind:

- leidliche Programmiererfahrung
- Kenntnisse in C ([KW: C])
- Kenntnisse im Umgang mit dem eigenen Betriebssystem

Wenn diese Voraussetzungen erfüllt sind, und das vorliegende Skript trotzdem unverständlich ist, dann bitte bei Klaus Wachtler beschweren; ebenso weitere

Fehlermeldungen, Verbesserungsvorschläge, Rückfragen zur weitergehenden Benutzung etc. an mich oder gleich nach `/dev/null`.

Verwendete **Warenzeichen** ohne weitere Kennzeichnung sind unter anderem:

- AutoCAD, ARX (AutoDesk)
- Microsoft, Visual Basic, Visual C++, Internet Explorer (Mickey Soft)
- Matlab (Mathworks)
- Netscape (Netscape Communications Corporation)
- Linux (Linus Torvalds)
- AnyWare (Klaus Wachtler)
- Postscript (Adobe Inc.)
- VMS (ehemals Digital Equipment Corporation/DEC, jetzt Compaq)

# 2 Einleitung

Einzelne wiederverwendbare Programmteile werden auf allen gängigen Systemen in Funktionsbibliotheken ausgelagert, wenn sie eine gewisse Allgemeingültigkeit haben und ausgetestet sind.

Beispielsweise die Standardfunktionen jeder Programmiersprache werden nicht mit jedem Programm mit kompiliert, sondern vorher einmalig in einer Bibliothek abgelegt und zu jedem Programm nur gelinkt.

Die ursprüngliche Vorgehensweise ist das sogenannte *statische Linken*: Zu jedem Programm werden zumindest die Teile der Bibliotheken hinzugefügt, die von dem jeweiligen Programm verwendet werden.

Dies führt aber zu einer immensen Redundanz: Wenn auf einem Unixsystem 98% aller Programme in C geschrieben sind, und fast alle die Funktionen `printf()` und `strcpy()` (neben vielen anderen) benötigen, dann sind diese Funktionen in mehrtausendfacher Kopie auf jedem Rechner vorhanden. Es wird aber nicht nur Plattenplatz verschwendet, sondern auch Speicherplatz zur Laufzeit: die mehrfach vorhandenen Funktionen der gleichzeitig im Speicher befindlichen Prozesse sind eigentlich überflüssig, weil sie stückweise identische Information, nämlich gleichen Maschinencode, enthalten. Solange die Programme den Maschinencode nicht verändern (was die meisten Systeme ohnehin unterbinden) reicht es, wenn der gemeinsam genutzte Maschinencode (ebenso wie konstanter Speicher) nur einmal im physikalischen Speicher vorliegt, und in den virtuellen Adreßraum aller Prozesse eingeblendet wird, die den Code benötigen.

Dazu wurden (zuerst für Multitaskingmaschinen wie VMS, Unix etc.) Konzepte zur mehrfachen Verwendung von Bibliotheken entworfen: das *dynamische Linken*.

In der Unix-Welt heißen solche Bibliotheken *shared objects*, abgekürzt *so*. Unter Windows heißt das Gegenstück *dynamic link library* oder *DLL*.

Das unter Linux praktizierte Vorgehen wurde von Solaris entlehnt und ist typisch für moderne Unixversionen. Unter Windows wird ein sehr ähnlicher Mechanismus bereitgestellt, der natürlich für alle Dinge ganz unterschiedliche Namen einführt; schließlich wurde das Konzept ja von Microsoft neu erfunden.

Abgesehen von etlichen Unterschieden im praktischen Betrieb (siehe **6 Wie eine dynamische Bibliothek gefunden wird** auf Seite 24 und **7 Versionen einer dynamischen Bibliothek** auf Seite 25) und einer willkürlich anderen Namensgebung ist die Verwendung unter Windows einerseits und Linux/Unix andererseits sehr ähnlich.

Die Beispiele werden für Linux anhand von `g++` (getestet mit Kernelversion 2.4.0 und `g++` Version 2.95.2) und für Windows (W2000, Visual C++ 6.0) vorgestellt.

Auf anderen Unix-Systemen gibt es mehr oder weniger ähnliche Mechanismen, die hier nicht behandelt werden (wer braucht noch Unix, seit es Linux gibt?).

## 2.1 Festes Linken oder dynamisches Laden

Auf beiden Systemen hat man zwei Möglichkeiten, um Funktionen (oder vollkommen analog: globale Variablen) einer dynamischen Bibliothek zu nutzen:

1. Anstelle der Funktionen (oder Daten) der Bibliothek werden beim Linken leere Dummy-Objekte (sogenannte *stubs*) dazugelinkt, um für den Linker die Referenzen zu erfüllen. Diese stubs machen nichts, aber bringen jeweils den Namen einer Funktion in die Objektdatei.

   Beim Start des Programms wird dann vom Laufzeitsystem automatisch die Bibliothek geladen (falls sie sich nicht schon durch einen anderen Prozeß im Speicher befindet), und die Verweise auf die Dummyfunktionen werden im Zuge des Relozierens durch Verweise auf die Funktionen der Bibliothek ersetzt. Wenn die Bibliothek wiederum unbefriedigte Referenzen enthält, werden ggf. weitere Bibliotheken nachgeladen. Dann startet das Programm wie üblich.

   Aus der Sicht des Aufrufers ändert sich nichts gegenüber statisch gelinkten Funktionen. Es findet durch den Compiler wie bei anderen Funktionen auch eine statische Typkontrolle statt.

   Die Bibliothek wird also vor dem tatsächlichen Start des Programms geladen und in den virtuellen Speicher des Programms eingeblendet, und verhält sich zur Laufzeit bezüglich der verschiedenen Speichersegmente (globale Variablen, Konstanten etc.) praktisch wie eine statisch gelinkte Bibliothek; siehe dazu auch **2.3 Speicherbereiche eines Prozesses** auf Seite 6.

Diese Verwendungsart wird im folgenden *festes Linken* genannt.

Genutzt wird der Mechanismus inzwischen für praktisch alle Laufzeitbibliotheken, also für Funktionen die sich selten ändern.

Gegenüber dem statischen Linken hat man neben dem geringeren Platten- und Speicherbedarf noch den Vorteil, daß die Bibliotheken durch neuere Versionen ersetzt werden können, ohne alle Programme kompilieren zu müssen. Konkretes Beispiel: Wenn Microsoft feststellt, daß in der Socketschnittstelle doch zuviele peinliche Fehler sind, dann wird mit dem nächsten Servicepack einfach die wsock32.dll ausgetauscht, und schon sind die alten Fehler durch neue ersetzt, ohne daß jemand auch nur ein vorhandenes Programm kompilieren oder linken muß.

Unter Windows entstehen beim Erzeugen der DLL zwei wichtige Dateien:

(a) eine Bibliotheksdatei irgendwas.lib, welche die Dummyfunktionen zum Linken der Programme enthält, und

(b) die eigentliche DLL irgendwas.dll, die den richtigen Programmcode enthält, und die beim Starten der Programme geladen wird.

Es gibt keinen Mechanismus, der sicherstellt daß zusammengehörige LIB- und DLL-Dateien verwendet werden. Man muß nur optimistisch sein.

Unter Unix dagegen gibt es für jedes *shared object* nur eine Datei; üblicherweise mit einem Namen der Art libirgendwas.so. Aus dieser Datei werden sowohl beim Linken der Programme die stubs generiert, als auch beim Starten der Programme der in den Speicher eingeblendete Maschinencode kopiert.

2. Man kann die Funktionen der Bibliothek aufrufen, indem man sie NICHT dem Compiler und damit dem Linker bekannt macht, sondern erst nach dem Programmstart mithilfe von Betriebssystemfunktionen die Bibliothek anhand ihres Dateinamens in den Speicher laden läßt (falls dies nicht bereits durch andere Prozesse geschehen ist) und ein *handle* für die geladene Bibliothek erhält. Dies ist i.d.R. ein Zeiger oder eine andere ganze Zahl als Kennung für die geladene Bibliothek.

   Mithilfe dieser Kennung kann man dann mit weiteren Systemaufrufen Zeiger auf die gewünschten Funktionen beschaffen (anhand der Funktionsnamen, die als String übergeben werden). Anhand der Funktionszeiger können dann die Funktionen aufgerufen werden.

   Da solche Funktionsaufrufe vollkommen der Kontrolle von Compiler und Linker entzogen sind, muß der Programmierer sicherstellen, daß Übergabeparameter und Rückgabewerte stimmen!

   Da sich insoweit zwischen Windows und Unix nur die Namensgebung unterscheidet, folgt hier gleich eine Gegenüberstellung der zugehörigen Stichworte:

| | Win32: | Unix: |
|---|---|---|
| Name des Mechanismus: | DLL (dynamic link library) | shared object |
| Typ für Handle der Bibl.: | `HINSTANCE` | `void*` |
| Funktion zum Laden: | `LoadLibrary()` | `dlopen()` |
| Funktion zum Entladen: | `FreeLibrary()` | `dlclose()` |
| Adresse zu suchen mit: | `GetProcAddress()` | `dlsym()` |
| Vereinbarungen in: | windows.h, winbase.h | dlfcn.h |

Dieser Mechanismus wird genutzt, um sehr flexibel Programmerweiterungen zu ermöglichen. Viele komplexe Programme lassen sich so nachträglich erweitern:

- AutoCAD kann mit sogenannten ARX-Dateien (*AutoCAD runtime extension*) erweitert werden; das sind DLLs, deren Namen allerdings nicht auf .DLL, sondern auf .ARX enden
- ein *plugin* für Netscape, Internet Explorer, Gimp, und so weiter ist genau eine solche dynamische Bibliothek
- Ein *ActiveX Control* (je nach Jahreszeit auch *OCX*, *COM-Objekt* oder so ähnlich genannt) basiert ebenfalls auf diesem Mechanismus (allerdings teilweise formal nicht als Bibliothek, sondern als EXE-Datei).
- Matlab kann auf allen unterstützten Plattformen (etliche Unixe, Linux, Win32) mit einem identischen Quelltext um neue Funktionen erweitert werden; die Betriebssystemabhängigkeiten werden von einer speziellen (plattformabhängigen) Bibliothek verdeckt.
- Sogar Visual Basic kann durch den Aufruf von DLLs an Geschwindigkeit zulegen, wenn man die relevanten Funktionen in C schreibt und als DLL nachlädt.

Meist stellen die aufrufenden Programme weitere Bedingungen an die geladene Bibliothek, beispielsweise eine aufzurufende Funktion mit einem definierten Verhalten.

Tatsächlich müssen der Name der Bibliothek, die Namen und die Parameter aber erst zur Laufzeit festliegen, im Extremfall durch Benutzereingabe.

Da solche Erweiterungen im Adreßraum des Aufrufers ausgeführt werden, sind sie einerseits so effektiv wie von vornherein eingebaute Funktionen, aber andererseits können sie bei Programmierfehlern auch ebenso leicht den gesamten Prozeß zum Absturz bringen.

**Warnung:** Aus demselben Grund haben nachgeladene Laufzeitbibliotheken die selben Rechte wie das aufrufende Programm. Wer also im Browser (Internet Explorer, Netscape etc.) ein Plugin laufen läßt, lebt gefährlich, wenn der Hersteller keinen Quelltext zur Verfügung stellt anhand dessen die Funktion des Plugin eingesehen werden kann!

Dasselbe gilt auch für COM/OLE/ActiveX-Objekte und ähnliche! Wer schön brav alle Attachments in emails löscht, ohne sie auszuführen, holt sich seine Feinde halt stattdessen mit einem Word-Dokument mit eingebetteten Objekten oder durch Laden eines Plugins für den Internet Explorer auf den Rechner.

In beiden Fällen sollte ein Debuggen unproblematisch sein, sofern das aufrufende Programm ebenso wie die Bibliothek mit den entsprechenden Compileroptionen übersetzt wurden.

## 2.2 Einsprungpunkte beim Laden oder Entladen einer dynamischen Bibliothek

Sowohl unter Windows als auch unter Linux kann man bei Bedarf einen Einsprungpunkt vorsehen, der beim Einblenden in den virtuellen Speicher beziehungsweise beim Entladen aufgerufen wird; beispielsweise um Startdaten zu lesen oder Resourcen zu allokieren oder freizugeben.

Ist die entsprechende Funktion nicht vorhanden, dann macht das nichts; sie wird dann natürlich auch nicht aufgerufen.

Unter Windows kann man dafür eine Funktion `DllMain()` definieren, die vom Lader des Betriebssystems mit einem Parameter aufgerufen wird; anhand dieses Parameters kann die Funktion erkennen, ob die DLL geladen oder entladen wird, oder ob eine zugehörige Task gestartet oder beendet wird. Siehe **3.2 Bibliothek Windowsversion** auf Seite 10.

Unter Linux kann man eine Funktion `_init()` definieren, die (falls vorhanden) beim Einblenden in den virtuellen Speicher ohne Parameter aufgerufen wird, und eine Funktion `_fini()`, die beim Entladen aufgerufen wird. Siehe dazu **3.1 Bibliothek Linuxversion** auf Seite 8. Die Verwendung dieser Einsprungpunkte setzt allerdings beim Übersetzen der Bibliothek die Option `-nostartfiles` voraus, was wiederum ein Übergeben von Ausnahmen über die Schnittstelle verhindert. Das ist aber keine große Einschränkung: Ausnahmen hat man nur in C++, und da kann man sich ein globales Objekt definieren, dessen Konstruktor und Destruktor die Aufgabe von `_init()` beziehungsweise `_fini()` übernehmen.

## 2.3 Speicherbereiche eines Prozesses

Beim Laden einer dynamischen Bibliothek werden die nichtkonstanten Datenbereiche (initialisierte Daten beispielsweise im `.data`-Segment; nicht initialisierte im `.bss`-Segment) der Bibliothek den entsprechenden Bereichen des aufrufenden Prozesses angegliedert; (nicht konstante) globale Variablen sind also so oft vorhanden wie die Bibliothek gerade verwendet wird: im Adreßbereich jedes Aufrufers einmal. Dies kann man an den Beispielprogrammen gut sehen, wenn man gleichzeitig mehrere Programme startet: die globale Variable `Wert`, deren Inhalt als Rückgabewert von `immereinsmehr()` (jeweils um eins erhöht) verwendet wird, beginnt auch bei mehreren Instanzen für jeden Prozeß neu zu zählen.

Konstante globale Daten (also beispielsweise das `.rodata` Segment) werden vom Betriebssystem ähnlich wie der Maschinencode nur einmal für alle Prozesse im realen Speicher angelegt, und in alle beteiligten Prozesse eingeblendet (allerdings gleichzeitig über die *memory management unit* (MMU) vor einem Beschreiben durch die Prozesse geschützt).

Als Stackspeicher nutzt jede Instanz einer dynamischen Bibliothek den Stack des aufrufenden Prozesses; anders könnte man ja auch gar keine Werte an die Funktionen der Bibliothek übergeben.

Beim Absturz eines der Prozesse wird dessen Speicher wie üblich freigegeben, und die Bibliothek verbleibt im Speicher zumindest bis der letzte nutzende Prozeß beendet ist.

Durch diese Maßnahmen erscheint jede Instanz einer dynamischen Bibliothek für den Aufrufer so, als ob er der einzige Nutzer der Bibliothek wäre. Erklärtes Ziel ist die saubere Trennung der Prozesse.

Insbesondere ist es nicht möglich, daß Prozesse über eine dynamische Bibliothek Daten austauschen (in der Art: ein Prozeß beschreibt eine Variable, ein anderer liest den geänderten Wert). Zur Kommunikation zwischen Prozessen sind also andere Mechanismen nötig.

**Tip Linux:** Wenn die globalen Symbole eines aufrufenden Programms verwendet werden sollen, um die Referenzen im *shared object* aufzulösen, dann muß beim Linken des Aufrufers die Option `-rdynamic` angegeben werden. Ansonsten sieht die Bibliothek die globalen Symbole des Aufrufers nicht, was auch erwünscht sein kann. Siehe `man 3 dlopen`.

Unter Windows hat man diese Wahlmöglichkeit nicht; hier sind die globalen Symbole des Aufrufers immer für die DLL sichtbar.

Gegebenenfalls kann man die Sichtbarkeit von globalen Objekten mit `static` steuern; siehe [KW: C].

# 3 Beispiel einer dynamisch ladbaren Bibliothek

Für die folgenden Beispiele wird eine kleine Bibliothek verwendet, die zwei Funktionen enthält:

1. Die Funktion `int summe( int a, int b )` bekommt zwei Argumente, und liefert die Summe der beiden.

2. Dagegen bekommt `int immereinsmehr( void )` nichts übergeben; beim ersten Aufruf liefert die Funktion den Wert 0, und bei den folgenden Aufruf jeweils einen um 1 höheren Wert.

**Tip C++**: Ein C++-Compiler erzeugt aus den Funktionsnamen, die der Programmierer vorsieht, und einer Kodierung der Parameterliste ziemlich kryptische tatsächliche Funktionsnamen (*name mangling*), um ein Überladen von Funktionen mit verschiedenen Parameterlisten (*Signaturen*) zu ermöglichen. Dies ist nett gemeint, aber sehr ärgerlich wenn man die Bibliothek zur Laufzeit dynamisch nachladen möchte. Denn um die Zeiger auf die Funktionen zu bekommen, muß man ihre vollen Namen wissen.

Die Funktion `int summe( int a, int b )` (ohne `extern "C"`) wird vom `g++` unter Linux beispielsweise mit dem Namen `summe__Fii` behandelt; der Visual C++ unter Windows macht daraus ein leicht zu merkendes `?summe@@YAHHH@Z`.

Deshalb ist es oft handlicher, die Schnittstelle zwischen Aufrufer und der Bibliothek als `extern "C"` zu deklarieren. Dann verwenden Compiler und Linker die unter C übliche Namensgebung; die vom Programmierer vergebenen Namen werden also beibehalten.

So wird auch in diesen Beispielen verfahren.

Trotz des Umsetzens des Aufrufs auf C-Konventionen ist es aber nach wie vor möglich, ober- und unterhalb der Schnittstelle C++ zu verwenden. Insbesondere können in einer Bibliothek Ausnahmen geworfen werden, die im aufrufenden Programm gefangen werden. (Das funktioniert sogar unter Windows.)

Falls man die Funktionen nutzen will, die als Einsprungpunkte dienen (`_init()` und `_fini()` unter Linux, oder `DllMain()` unter Windows) dann **müssen** diese nach C-Konvention benannt sein; wenn man seine Bibliothek in C++ schreibt, muß man also diese Funktionen als `extern "C"` deklarieren.

## 3.1 Bibliothek Linuxversion

Neben den beiden eigentlich genutzten Funktionen sind noch die Funktionen `_init()` und `_fini()` enthalten, die beim Laden beziehungsweise Entladen der DLL aufgerufen werden (siehe **2.2 Einsprungpunkte beim Laden oder Entladen einer dynamischen Bibliothek** auf Seite 6).

```
///////////////////////////////////////////////////////////////////////
//
// Time-stamp: "(09.05.01 20:41) demodll.cpp [Klaus Wachtler]"
//
//
// Kompilieren mit:
//  g++ -shared demodll.cpp -o libdemodll.so
// (falls Ausnahmen gewünscht; dann sind _init() und _fini()
// auszukommentieren), oder
//  g++ -shared -nostartfiles demodll.cpp -o libdemodll.so
```

```
// (_init() und _fini() möglich, aber keine Ausnahmen).
//

static int    Wert = 0;

extern "C" int summe( int a, int b )
{
  return a + b;
}

extern "C" int immereinsmehr( void )
{
  if( Wert>6 )
    {
      throw( 125 );
    }

  return Wert++;
}

// Ab hier kann der ganze Rest entfallen, wenn man ihn nicht braucht:

//#include <stdio.h>
//
//extern "C" void _init( void )
//{
//  printf( "Bibliothek geladen\n" );
//}
//
//extern "C" void _fini( void )
//{
//  printf( "Bibliothek beendet\n" );
//}


//
///////////////////////////////////////////////////////////////////////
```

Die Beispielbibliothek kann folgendermaßen übersetzt werden:

```
klaus@aw33:~/db/linux > g++ -Wall -shared demodll.cpp -o libdemodll.so
```

Dadurch erhält man das *shared object* libdemodll.so.

## 3.2 Bibliothek Windowsversion

Neben den beiden eigentlich genutzten Funktionen ist noch die Funktion `DllMain()` enthalten, die beim Laden und Entladen der DLL aufgerufen wird (siehe **2.2 Einsprungpunkte beim Laden oder Entladen einer dynamischen Bibliothek** auf Seite 6).

Damit ein Aufrufer irgendwelche globalen Namen (wie hier die Funktionsnamen) ansprechen kann, müssen diese mit `__declspec(dllexport)` deklariert werden.

```
///////////////////////////////////////////////////////////////////////
//
// Time-stamp: "(09.05.01 20:00) demodll.cpp [Klaus Wachtler]"
//
//
// Kompilieren mit:
//  cl -LD -GX demodll.cpp
//

static int    Wert = 0;

extern "C" __declspec(dllexport) int summe( int a, int b )
{
  return a + b;
}

extern "C" __declspec(dllexport) int immereinsmehr( void )
{
  if( Wert>6 )
    {
      throw( 125 );
    }

  return Wert++;
}

// Ab hier kann der ganze Rest entfallen, wenn man ihn nicht braucht:

#include <stdio.h>
#include <windows.h>
#include <winbase.h>


// ModulHandle ist das "module handle" resp. "instance handle".
BOOL APIENTRY DllMain( HANDLE ModulHandle,
       DWORD  InitOderExit,
       LPVOID lpReserved
       )
```

```
{
  switch( InitOderExit )
    {
    case DLL_PROCESS_ATTACH:
      // Die DLL wird in den Adreßraum eines Prozesses eingeblendet
      // (z.B. durch ein LoadLibrary() bei explizitem Laden, oder
      // durch Programmstart bei implizitem Linken über DEF-Dateien).
      // Entspricht dem Starten eines normalen Programms.
      // Guter Moment, um globale Variablen zu initialisieren etc.

      fprintf( stderr, "DllMain: DLL_PROCESS_ATTACH\n" );

      break;

    case DLL_THREAD_ATTACH:
      // Der zugehörige Prozeß hat einen neuen thread gestartet.
      // Hier könnte man thrad local storage allokieren (TlsAlloc())

      fprintf( stderr, "DllMain: DLL_THREAD_ATTACH\n" );

      break;

    case DLL_THREAD_DETACH:
      // Der zugehörige Prozeß hat einen thread gestoppt.
      // Gegenstück zu DLL_THREAD_ATTACH.

      fprintf( stderr, "DllMain: DLL_THREAD_DETACH\n" );

      break;

    case DLL_PROCESS_DETACH:
      // Die DLL wird aus dem Prozeßraum eines Prozesses entfernt
      // (durch Ende des Prozesses oder durch FreeLibrary()).
      // Gegenstück zu DLL_PROCESS_ATTACH.
      // Hier kann man globale Objekte wieder aufräumen.

      fprintf( stderr, "DllMain: DLL_PROCESS_DETACH\n" );

      break;

    default:
      // sollte eigentlich nie vorkommen!

      fprintf( stderr, "DllMain: DLL_???\n" );

      break;
    }
```

```
  // Rückgabewert von DllMain() sollte !=(0) sein, wenn alles ok ist.
  // Die DLL gilt dann als erfolgreich geladen.
  // Im Fehlerfall sollte DllMain() (0) liefern.
  // Dann liefert LoadLibrary() dem aufrufenden Prozeß einen Wert
  // ungleich 0 (also "gescheitert) bei explizitem Linken, bzw.
  // bei implizitem Linken wird der gerade startende Prozeß
  // abgeschossen.

  return TRUE;
}

//
///////////////////////////////////////////////////////////////////////
```

Die Beispielbibliothek kann folgendermaßen übersetzt werden:

```
C:\temp\dynBib\win32>cl -LD -GX demodll.cpp
Optimierender Microsoft (R) 32-Bit C/C++-Compiler, Version 12.00.8168, fuer x86
Copyright (C) Microsoft Corp 1984-1998. Alle Rechte vorbehalten.

demodll.cpp
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/out:demodll.dll
/dll
/implib:demodll.lib
demodll.obj
   Bibliothek demodll.lib und Objekt demodll.exp wird erstellt
```

Die wesentlichen daraus entstehenden Dateien sind demodll.lib und demodll.dll.

Die LIB-Datei wird benötigt, um die Bibliothek fest an ein Programm zu linken.

Die DLL wird sowohl zum festen Linken als auch zum dynamischen Nachladen nach Programmstart benötigt.

# 4 Beispielprogramm zum festen Linken

Für Windows und Linux folgt ein Beispielprogramm, das fest an die in **3 Beispiel einer dynamisch ladbaren Bibliothek** auf Seite 7 gezeigte Bibliothek gebunden wird. Die Funktionen werden also wie normale Unterprogramme aufgerufen.

Die aufgerufenen Funktionen müssen entsprechend ihrer Definition in der Bibliothek deklariert werden.

## 4.1 Linuxversion

```
////////////////////////////////////////////////////////////////////////
//
// Time-stamp: "(09.05.01 07:33) linkaufrufer.cpp [Klaus Wachtler]"
//
// kleine Demo zum Laden und Verwenden einer DLL.
//
// Kompilieren und Linken:
//    g++ linkaufrufer.cpp libdemodll.so -o linkaufrufer

#include <stdio.h>
#include <unistd.h>

// Deklarationen der Funktionen in der SO:
extern "C" int summe( int, int );
extern "C" int immereinsmehr( void );

int main()
{
  // Aufruf der Funktionen:
  for( int i=0; i<10; i++ )
    {
      try
        {
          printf( "Summe von %d und %d ist %d; immer eins mehr ist %d\n",
                  i,
                  i/2,
                  summe( i, i/2 ),
                  immereinsmehr()
                  );
        }
      catch( int Fehler )
        {
          fprintf( stderr, "int-Ausnahme %d gefangen!\n", Fehler );
        }
      catch( ... )
        {
          fprintf( stderr, "irgendeine Ausnahme gefangen!\n" );
        }
      sleep( 1 );
    }

  return 0;
}

//
////////////////////////////////////////////////////////////////////////
```

Das Beispielprogramm kann folgendermaßen übersetzt und gestartet werden:

```
klaus@aw33:~/db/linux > unset LD_LIBRARY_PATH
klaus@aw33:~/db/linux > g++ -Wall linkaufrufer.cpp libdemodll.so -o linkaufrufer
klaus@aw33:~/db/linux > linkaufrufer
linkaufrufer: error in loading shared libraries: libdemodll.so:
 cannot open shared object file: No such file or directory
klaus@aw33:~/db/linux > export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
klaus@aw33:~/db/linux > linkaufrufer
Summe von 0 und 0 ist 0; immer eins mehr ist 0
Summe von 1 und 0 ist 1; immer eins mehr ist 1
Summe von 2 und 1 ist 3; immer eins mehr ist 2
Summe von 3 und 1 ist 4; immer eins mehr ist 3
Summe von 4 und 2 ist 6; immer eins mehr ist 4
Summe von 5 und 2 ist 7; immer eins mehr ist 5
Summe von 6 und 3 ist 9; immer eins mehr ist 6
int-Ausnahme 125 gefangen!
int-Ausnahme 125 gefangen!
int-Ausnahme 125 gefangen!
```

Die Beispielkommandos und ihre Ausgabe zeigen gleichzeitig die Verwendung der Umgebungsvariablen `LD_LIBRARY_PATH`. Wenn diese gesetzt ist, wird in den darin angegebenen (und mit einem Doppelpunkt getrennten) Pfaden nach dynamischen Bibliotheken gesucht.

**Tip:** Welche dazugelinkten dynamischen Bibliotheken von einem Programm (oder einer Bibliothek wiederum) benötigt werden, kann man unter Linux mit dem Kommando `ldd` erfahren.

Dies zeigt aber nur die direkt benötigten Bibliotheken an; davon wiederum abhängige muß man gegebenenfalls mit weiteren Aufrufen ermitteln.

Zur Laufzeit nachgeladene Bibliotheken (siehe **5 Beispiel zum Laden einer Bibliothek zur Laufzeit** auf Seite 16) werden so allerdings nicht ermittelt.

## 4.2 Windowsversion

```
// Time-stamp: "(09.05.01 05:51) linkaufrufer.cpp [Klaus Wachtler]"
//
// kleine Demo zum Laden und Verwenden einer DLL.
//
// Kompilieren und Linken:
//    cl -GX linkaufrufer.cpp demodll.lib
// Zum Linken ist die Datei demodll.lib nötig.
// Zum Starten muß die demodll.dll im gleichen Verzeichnis sein
// (oder sonstwie erreichbar -> PATH-Variable).
```

```
#include <stdio.h>
#include <windows.h>
#include <winbase.h>

// Deklarationen der Funktionen in der DLL:
extern "C" int summe( int, int );
extern "C" int immereinsmehr( void );

int main()
{
  // Aufruf der Funktionen:
  for( int i=0; i<10; i++ )
    {
      try
        {
          printf( "Summe von %d und %d ist %d; immer eins mehr ist %d\n",
                  i,
                  i/2,
                  summe( i, i/2 ),
                  immereinsmehr()
                  );
        }
      catch( int Fehler )
        {
          fprintf( stderr, "int-Ausnahme %d gefangen!\n", Fehler );
        }
      catch( ... )
        {
          fprintf( stderr, "irgendeine Ausnahme gefangen!\n" );
        }
      Sleep( 1000 );
    }

  return 0;
}
```

Das Beispielprogramm kann folgendermaßen übersetzt und gestartet werden:

```
C:\temp\dynBib\win32> cl -GX linkaufrufer.cpp demodll.lib
Optimierender Microsoft (R) 32-Bit C/C++-Compiler, Version 12.00.8168, fuer x86
Copyright (C) Microsoft Corp 1984-1998. Alle Rechte vorbehalten.

linkaufrufer.cpp
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/out:linkaufrufer.exe
```

```
linkaufrufer.obj
demodll.lib

C:\temp\dynBib\win32>linkaufrufer
DllMain: DLL_PROCESS_ATTACH
Summe von 0 und 0 ist 0; immer eins mehr ist 0
Summe von 1 und 0 ist 1; immer eins mehr ist 1
Summe von 2 und 1 ist 3; immer eins mehr ist 2
Summe von 3 und 1 ist 4; immer eins mehr ist 3
Summe von 4 und 2 ist 6; immer eins mehr ist 4
Summe von 5 und 2 ist 7; immer eins mehr ist 5
Summe von 6 und 3 ist 9; immer eins mehr ist 6
int-Ausnahme 125 gefangen!
int-Ausnahme 125 gefangen!
int-Ausnahme 125 gefangen!
DllMain: DLL_PROCESS_DETACH
```

**Tip:** Welche dazugelinkten dynamischen Bibliotheken von einem Programm (oder einer Bibliothek wiederum) benötigt werden, kann man unter Windows mit dem Kommando `DUMPBIN /imports` erfahren.

Dies zeigt aber nur die direkt benötigten Bibliotheken an; davon wiederum abhängige muß man gegebenenfalls mit weiteren Aufrufen ermitteln.

Dafür gibt es ein anderes Programm `Dependancy Walker` (von Steve P. Miller)[1], das einigen Entwicklungspaketen von Microsoft beiliegt.

Zur Laufzeit nachgeladene Bibliotheken (siehe **5 Beispiel zum Laden einer Bibliothek zur Laufzeit** auf Seite 16) werden so allerdings auch nicht ermittelt.

# 5 Beispiel zum Laden einer Bibliothek zur Laufzeit

Für Windows und Linux folgt ein Beispielprogramm, das die in **3 Beispiel einer dynamisch ladbaren Bibliothek** auf Seite 7 gezeigte Bibliothek nach Programmstart lädt und darin enthaltene Funktionen über Zeiger aufruft.

Die Zeiger auf die aufzurufenden Funktionen müssen entsprechend der Definition in der Bibliothek deklariert werden.

Nach dem einmaligen Laden der dynamischen Bibliothek können beliebig oft Zeiger auf darin enthaltene Objekte (Funktionen, globale Variablen) geholt werden, und die Objekte dann über die Zeiger angesprochen werden.

[1] http://www.dependencywalker.com/

## 5.1 Linuxversion

Ein *shared object* wird mit der Funktion `dlopen()` geladen; bei Bedarf kann man sie mit `dlclose()` wieder freigeben.

Beim Laden wird hier der Parameter `RTLD_LAZY` angegeben. Das bedeutet, daß in der geladenen Bibliothek vorhandene Referenzen nicht sofort befriedigt werden sollen, sondern erst später beim erstmaligen Aufruf der einzelnen Funktionen (nicht auflösbare Referenzen führen auch erst dann zu Fehlern). Alternativ könnte man auch `RTLD_NOW` angeben; dann werden Referenzen sofort aufgelöst beziehungsweise als Fehler erkannt, wenn sie nicht auflösbar sind.

In beiden Fällen kann man den verwendeten Wert noch mit `RTLD_GLOBAL` verODERn (also beispielsweise `RTLD_NOW|RTLD_GLOBAL`); dann werden die globalen Symbole der jetzt geladenen Bibliothek zum Auflösen der Referenzen in **zukünftig geladenen** Bibliotheken verwendet.

Wenn man die globalen Symbole des aufrufenden Programms für die Referenzen der Bibliotheken verwenden möchtem dann muß das aufrufende Programme mit der Option `-rdynamic` gelinkt worden sein.

Wenn die geladene Bibliothek eine Funktion `_init()` enthält, dann wird diese aufgerufen, bevor `dlopen()` beendet ist.

`dlopen()` liefert im Erfolgsfall einen `void*`-Zeiger ungleich `NULL`, der im weiteren verwendet wird, um mit `dlsym()` Zeiger auf Funktionen (oder bei Bedarf auf globale Variablen) in der eingeblendeten Bibliothek zu finden.

Die ganzen `dl...()`-Funktionen haben einen gemeinsamen Fehlerindikator; jeweils der letzte Fehler kann (als String `const char*`) mit der Funktion `dlerror()` abgefragt werden. Allerdings wird von `dlerror()` der Fehlerindikator gelöscht; weitere Aufrufe liefern dann keine Fehlermeldung mehr (bis ein weiterer Fehler auftritt natürlich nur).

In dem folgenden Quelltext wird für die beiden aufgerufenen Funktionen jeweils ein Typ mit `typedef` definiert, um die Funktionszeiger optisch etwas ansprechender handhaben zu können.

Mit `dlsym()` holt man sich anhand des Namens eines Objekts (Funktion, globale Variable) eine Adresse in den eigenen Adreßraum (oder `NULL` im Fehlerfall).

Über den von `dlsym()` gelieferten Zeiger kann man die Objekte der Bibliothek ansprechen; in diesem Beispiel also die Funktionen aufrufen.

```
//////////////////////////////////////////////////////////////////////
//
// Time-stamp: "(27.05.01 19:30) dynaufrufer.cpp [Klaus Wachtler]"
//
```

```
// kleine Demo zum Laden und Verwenden eines shared object.
// Kompilieren und Linken:
//    g++ -rdynamic dynaufrufer.cpp -o dynaufrufer -ldl
// Zum Starten muß die libdemodll.so gefunden werden können;
// siehe man dlopen.

#include <stdio.h>
#include <unistd.h>
#include <dlfcn.h>


// Typ: Zeiger auf Funktion, die int liefert, zwei int erhält:
typedef int ( *int_intint_pf_t)( int, int );

// Typ: Zeiger auf Funktion, die int liefert, keine Parameter:
typedef int ( *int_void_pf_t)( void );



int main()
{
  // Handle für geladene Bibliothek:
  void  *so_handle = NULL;

  // Zeiger auf die Funktion in der Bibliothek:
  int_intint_pf_t summe_pf = NULL;
  int_void_pf_t   immereinsmehr_pf = NULL;

  // einmalig die Bibliothek laden:
  so_handle = dlopen( "libdemodll.so", RTLD_LAZY );

  if( !so_handle )
    {
      // Laden der Bibliothek hat nicht geklapppt!
      char *Fehlertext = dlerror();
      fprintf( stderr,
               "Fehler beim Laden: %s\n",
               ( Fehlertext ? Fehlertext : "??????" )
               );
      exit( 2 );
    }

  // Zeiger auf die darin enthaltenen Funktionen holen:
  summe_pf
    = (int_intint_pf_t)dlsym( so_handle, "summe" );
  immereinsmehr_pf
    = (int_void_pf_t)dlsym( so_handle, "immereinsmehr" );
```

```
// Funktion gefunden?
if( !summe_pf )
  {
    char *Fehlertext = dlerror();
    fprintf( stderr,
             "Fehler! Funktion summe() nicht gefunden (%s)\n",
             ( Fehlertext ? Fehlertext : "??????" )
             );
    exit( 2 );
  }
// Funktion gefunden?
if( !immereinsmehr_pf )
  {
    // Nö!
    char *Fehlertext = dlerror();
    fprintf( stderr,
             "Fehler! Funktion immereinsmehr() nicht gefunden (%s)\n",
             ( Fehlertext ? Fehlertext : "??????" )
             );
    exit( 2 );
  }

// und beliebig oft benutzen:
for( int i=0; i<10; i++ )
  {
    try
      {
        printf( "Summe von %d und %d ist %d; immer eins mehr ist %d\n",
                i,
                i/2,
                (*summe_pf)( i, i/2 ),
                (*immereinsmehr_pf)()
                );
      }
    catch( int Fehler )
      {
        fprintf( stderr, "int-Ausnahme %d gefangen!\n", Fehler );
      }
    catch( ... )
      {
        fprintf( stderr, "irgendeine Ausnahme gefangen!\n" );
      }
    sleep( 1 );
  }

// Mann kann die Bibliothek auch wieder entladen, wenn man sie nicht
// mehr braucht.
// Kann man sich aber auch sparen.
```

```
  dlclose( so_handle );

  return 0;
}

//
///////////////////////////////////////////////////////////////////////////
```

Das Beispielprogramm kann folgendermaßen übersetzt und gestartet werden:

```
klaus@aw33:~/db/linux > g++ -Wall -rdynamic dynaufrufer.cpp -o dynaufrufer -ldl
klaus@aw33:~/db/linux > ./dynaufrufer
Fehler beim Laden: libdemodll.so: cannot open shared object file:
 Datei oder Verzeichnis nicht gefunden
klaus@aw33:~/db/linux > export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
klaus@aw33:~/db/linux > ./dynaufrufer
Summe von 0 und 0 ist 0; immer eins mehr ist 0
Summe von 1 und 0 ist 1; immer eins mehr ist 1
Summe von 2 und 1 ist 3; immer eins mehr ist 2
Summe von 3 und 1 ist 4; immer eins mehr ist 3
Summe von 4 und 2 ist 6; immer eins mehr ist 4
Summe von 5 und 2 ist 7; immer eins mehr ist 5
Summe von 6 und 3 ist 9; immer eins mehr ist 6
int-Ausnahme 125 gefangen!
int-Ausnahme 125 gefangen!
int-Ausnahme 125 gefangen!
```

**Tips:**

- Wenn man wissen möchte, welche Bibliotheken dynamisch nachgeladen werden, dann kann man das mit `ltrace` erfahren. Damit erhält man unter anderem die Aufrufe von `dlopen()` protokolliert.

  Die eigentlichen Aufrufe der Funktionen in der Bibliothek sind allerdings nicht mehr zu protokollieren (zumindest nicht ohne Debugger), weil sie nur noch über Zeiger stattfinden.

- Hat man zu einer Bibliothek nicht den Quelltext, aber möchte trotzdem wissen, welche Symbole (globale Daten, Funktionsnamen) enthalten sind; dann kann `nm` verwenden. Dieses Kommando listet für Objektdateien, also auch für dynamische Bibliotheken, die darin vorkommenden Symbole.

  Namen, die in einer Objektdatei referenziert werden, aber nicht vorhanden sind, ist ein `U` vorangestellt; solche *unsatisfied externals* müssen beim Linken beziehungsweise beim Starten eines Programms durch weitere Bibliotheken befriedigt werden.

  Dagegen ist Symbolen, die in der aktuellen Objektdatei definiert sind und von anderen Objektdateien referenziert werden können, in der `nm`-Ausgabe ein T vorangestellt. So gekennzeichnete Funtionen sind Kandidaten für einen dlsym()-Aufruf.

## 5.2 Windowsversion

Das Laden der Bibliothek erfolgt mit einem Aufruf von LoadLibrary(); Ergebnis ist im Erfolgsfall vom Typ HINSTANCE (hinter dem sich wiederum ein void* verbirgt, welch ein Wunder!).

Freigeben kann man eine DLL mit FreeLibrary().

Mit GetProcAddress() kann man dann (analog zu dlsym() unter Linux) Zeiger auf die Objekte in der DLL holen.

```
///////////////////////////////////////////////////////////////////////
//
// Time-stamp: "(27.05.01 19:29) dynaufrufer.cpp [Klaus Wachtler]"
//
// kleine Demo zum Laden und Verwenden einer DLL.
// Kompilieren und Linken:
//    cl -GX dynaufrufer.cpp
// Zum Starten muß die demodll.dll im gleichen Verzeichnis sein
// (oder sonstwie erreichbar -> PATH-Variable).

#include <stdio.h>
#include <windows.h>
#include <winbase.h>

// Typ: Zeiger auf Funktion, die int liefert, zwei int erhält:
typedef int ( __stdcall *int_intint_pf_t)( int, int );

// Typ: Zeiger auf Funktion, die int liefert, keine Parameter:
typedef int ( __stdcall *int_void_pf_t)( void );



int main()
{
  // Handle für geladene DLL:
  HINSTANCE  DLL_Handle = NULL;

  // Zeiger auf die Funktion in der DLL:
  int_intint_pf_t summe_pf = NULL;
  int_void_pf_t   immereinsmehr_pf = NULL;

  // einmalig die Bibliothek laden:
```

```
DLL_Handle = LoadLibrary(  "demodll.dll");

if( !DLL_Handle )
  {
    // Laden dder DLL hat nicht geklapppt!
    // Mit GetLastError() könnte man jetzt eine Kennung des
    // aufgetretenen Fehlers beschaffen; was man mit dieser Kennung
    // dann anstellt habe ich nicht verstanden :-(
    fprintf( stderr, "Fehler beim Laden der DLL!\n" );
    exit( 2 );
  }


// Zeiger auf die darin enthaltenen Funktionen holen:
summe_pf
  = (int_intint_pf_t)GetProcAddress( DLL_Handle, "summe" );
immereinsmehr_pf
  = (int_void_pf_t)GetProcAddress( DLL_Handle, "immereinsmehr" );

// Funktion gefunden?
if( !summe_pf )
  {
    // Nö!
    fprintf( stderr,
             "Fehler! Funktion summe() nicht gefunden\n"
             );
    exit( 2 );
  }
// Funktion gefunden?
if( !immereinsmehr_pf )
  {
    fprintf( stderr,
             "Fehler! Funktion immereinsmehr() nicht gefunden\n"
             );
    exit( 2 );
  }

// und beliebig oft benutzen:
for( int i=0; i<10; i++ )
  {
    try
      {
        printf( "Summe von %d und %d ist %d; immer eins mehr ist %d\n",
                i,
                i/2,
                (*summe_pf)( i, i/2 ),
                (*immereinsmehr_pf)()
                );
```

```
        }
      catch( int Fehler )
        {
          fprintf( stderr, "int-Ausnahme %d gefangen!\n", Fehler );
        }
      catch( ... )
        {
          fprintf( stderr, "irgendeine Ausnahme gefangen!\n" );
        }
      Sleep( 1000 );
    }

  // Mann kann die DLL auch wieder entladen, wenn man sie nicht
  // mehr braucht (FreeLibrary()).
  // Kann man sich aber auch sparen.

  FreeLibrary( DLL_Handle );

  return 0;
}

//
////////////////////////////////////////////////////////////////////////
```

Das Beispielprogramm kann folgendermaßen übersetzt und gestartet werden (die Compileroption `-GX` ist nötig, weil Ausnahmen verwendet werden, was in C++ ja so ungewöhnlich ist!):

```
C:\temp\dynBib\win32>cl -GX dynaufrufer.cpp
Optimierender Microsoft (R) 32-Bit C/C++-Compiler, Version 12.00.8168, fuer x86
Copyright (C) Microsoft Corp 1984-1998. Alle Rechte vorbehalten.

dynaufrufer.cpp
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/out:dynaufrufer.exe
dynaufrufer.obj

C:\temp\dynBib\win32>dynaufrufer
DllMain: DLL_PROCESS_ATTACH
Summe von 0 und 0 ist 0; immer eins mehr ist 0
Summe von 1 und 0 ist 1; immer eins mehr ist 1
Summe von 2 und 1 ist 3; immer eins mehr ist 2
Summe von 3 und 1 ist 4; immer eins mehr ist 3
Summe von 4 und 2 ist 6; immer eins mehr ist 4
Summe von 5 und 2 ist 7; immer eins mehr ist 5
Summe von 6 und 3 ist 9; immer eins mehr ist 6
int-Ausnahme 125 gefangen!
```

```
int-Ausnahme 125 gefangen!
int-Ausnahme 125 gefangen!
DllMain: DLL_PROCESS_DETACH
```

Ein analoges Kommando zu `ltrace` gibt es meines Wissens unter Windows nicht.

# 6 Wie eine dynamische Bibliothek gefunden wird

Wo nach einer dynamischen Bibliothek gesucht wird, ist zum einen interessant, um ein Programm überhaupt starten zu können, und zum anderen wirkt sich die Suchreihenfolge darauf aus, welche Bibliothek verwendet wird, wenn mehrere Versionen mit dem selben Namen existieren.

Wenn beim Laden der Bibliothek der volle Pfadname angegeben wird, dann ist die Sache klar.

Ansonsten gilt:

**Unter Linux** wird zum Auflösen externer Referenzen das Programm ld.so verwendet, welches nach Bibliotheken in folgender Reihenfolge sucht (aus [Systemzentrale]):

- Alle Pfade, die in der *environment*-Variablen `LD_LIBRARY_PATH` angegeben sind (jeweils durch Doppelpunkt getrennt) (nicht für *setuid*- und *setguid*-Programme)
- Wenn beim Linken eines Programms die Option `-rpath` *pfad* angegeben war, dann dort
- In einem internen Cache des Kernels, in dem ständig benötigte Bibliotheken mit ihren Pfaden vorgehalten werden (siehe Anmerkungen unten zu `ldconfig`)
- /lib
- /usr/lib

Der interne Cache für die Bibliotheken wird vom Programm `ldconfig` aktualisiert. `ldconfig` liest aus /etc/ld.so.conf eine Liste von Verzeichnissen, sucht in allen diesen Verzeichnissen nach shared objects, und trägt diese dann in den internen Cache ein. `ldconfig` kann üblicherweise nur als Benutzer `root` gestartet werden.

**Unter Windows** wird nach einer DLL zuerst im Verzeichnis gesucht, aus dem die zugehörige EXE gestartet wurde, dann in den Verzeichnissen der `PATH`-Variable.

Soweit das von Microsoft dokumentierte Verhalten. Zudem kommt es aber manchmal vor, daß eine Bibliothek nicht erneut geladen wird, wenn ein anderes Programm eine DLL gleichen Namens bereits in den Speicher gebracht hat. Dann hängt das Verhalten des neu gestarteten Programms (nämlich welche DLL verwendet wird) davon ab, welche anderen Programme bereits laufen. Wenn das mal gut geht… (siehe dazu auch **7 Versionen einer dynamischen Bibliothek** auf Seite 25). Wie einheitlich sich hier die verschiedenen Windowsversionen verhalten, weiß ich nicht.

**Konkretes Beispiel:** Ein Küchenhersteller hatte unter Windows NT ein Serverprogramm, welches eine Kommunikation zwischen einer grafischen Auftragserfassung in AutoCAD einerseits und einer Warenwirtschaft auf einem Unixrechner andererseits realisiert. Das Serverprogramm benötigt natürlich die Datei wsock32.dll, in der die Funktionen zur Socketprogrammierung liegen.

Soweit funktioniert alles meistens prima, nur manchmal kann das Serverprogramm keine Verbindung zum Unixrechner aufbauen. Als Ursache stellte sich nun heraus, daß manchmal auch die BTX-Software von T-Online lief, die eine eigene Version der DLL lädt, und das Serverprogramm genau dann nicht mehr läuft, wenn es hinterher gestartet wird: es verwendet dann die DLL, die von T-Online mitgeliefert wird, und die hat nun leider irgendwo ein paar Fehler. Solche Abhängigkeiten zu finden ist reine Glückssache.

# 7 Versionen einer dynamischen Bibliothek

**Unter Windows** gibt es keinen echten Mechanismus, mehrere Versionen einer DLL auf einem Rechner zu halten, und verschiedenen Programmen gezielt bestimmte Versionen zuordnen zu können. Als Ausweg bleibt eigentlich nur, zu jedem Programm die passenden DLLs in das selbe Verzeichnis zu kopieren, und auf die richtige Suchreihenfolge seitens Windows zu vertrauen (siehe **6 Wie eine dynamische Bibliothek gefunden wird** auf Seite 24).

Damit wird zwar die Platzersparnis durch dynamische Bibliotheken verschenkt, aber was will man machen. Hier rächt sich halt das konzeptlose Umsetzen einer an sich guten Idee.

Eigentlich sollte das alles kein Problem sein, weil Microsoft verlangt, daß von einer DLL mit einem bestimmten Namen immer nur die neueste Version auf einem Rechner vorhanden sein soll, und diese Version alles kann, was alle älteren Versionen auch konnten – und zwar so kompatibel, daß Programme, die mit einer älteren LIB-Datei gelinkt wurden, keinen Unterschied erkennen können.

Bei jeder Installation einer DLL soll das Installationsprogramm eine ältere Version einfach überschreiben, und die alten Programm müssten ungestört weiter funktionieren.

(Soweit die Theorie.)

**Unter Unix und Linux** dagegen hat es sich eingebürgert, alle Bibliotheken, also auch *shared objects*, mit einer (meist) dreistufigen Versionsnummer zu versehen. Diese drei Zahlen werden einfach mit Punkten getrennt an den Dateinamen angehängt.

Die Bibliothek mit den Funktionen aus awutil.h, also libawutil.so, heißt also in Wirklichkeit libawutil.so.1.0.0, wenn sie in der ersten Version vorliegt.

Der eigentliche Name der Bibliothek (ohne Versionsnummern, also etwa libawutil.so) wird dann als symbolischer Link (siehe `man 1 ln`) auf die höchste existierende Hauptversion (libawutil.so.1) angelegt. Diese Datei ist wiederum nur ein symbolischer Link auf ihre zugehörige höchste Unterversion (libawutil.so.1.0); diese ist dann analog ein symbolischer Link auf ihre höchste Unterversion (libawutil.so.1.0.0).

Bei kleineren Änderungen (beispielsweise reine Fehlerkorrekturen) ändert man nur die letzte Nummer, und installiert also bei mehreren Korrekturen nacheinander die Versionen libawutil.so.1.0.1, libawutil.so.1.0.2 und libawutil.so.1.0.3. Der symbolische Link libawutil.so.1.0 wird dabei jeweils angepaßt, so daß neu gelinkte Programme immer die aktuelle Version bekommen.

Wenn man *Erweiterungen* der Bibliothek vornimmt, die rückwärtskompatibel sind, dann erhöht man die mittlere Zahl. Die zugehörigen Fehlerkorrekturen drücken sich dann wieder in der letzten Zahl aus. Beispielsweise würde man dann nacheinander die Versionen libawutil.so.1.1.0, libawutil.so.1.1.1 und libawutil.so.1.1.2 installieren.

Dabei sind wieder die symbolischen Links anzupassen.

Auf einem konkreten Rechner könnte das anfangs so aussehen:

```
klaus@aw33:~ > ls -l /usr/local/lib/
...
lrwxrwxrwx 1 root root     14 Sep 29 2000 libawutil.so -> libawutil.so.1
lrwxrwxrwx 1 root root     16 Sep 29 2000 libawutil.so.1 -> libawutil.so.1.0
lrwxrwxrwx 1 root root     18 Sep 29 2000 libawutil.so.1.0 -> libawutil.so.1.0.0
-rwxr-xr-x 1 root root 110500 Sep 29 2000 libawutil.so.1.0.0
...
```

Wenn ein Programm mit einer der Bibliotheken libawutil.so, libawutil.so.1, libawutil.so.1.0 oder libawutil.so.1.0.0 gelinkt wird, dann wird in allen drei Fällen effektiv dieselbe Bibliothek verwendet. Jedes so gelinkte Programm wird dementsprechend nach dem Starten Funktionen aus libawutil.so.1.0.0 aufrufen können.

Installiert man jetzt eine fehlerbereinigte Version libawutil.so.1.0.1 und danach libawutil.so.1.0.2, dann paßt man die symbolischen Links entsprechend an; das Inhaltsverzeichnis könnte dann etwa so aussehen:

```
klaus@aw33:~ > ls -l /usr/local/lib/
...
lrwxrwxrwx 1 root root     14 Sep 29 2000 libawutil.so -> libawutil.so.1
lrwxrwxrwx 1 root root     16 Sep 29 2000 libawutil.so.1 -> libawutil.so.1.0
lrwxrwxrwx 1 root root     18 Dec 24 2000 libawutil.so.1.0 -> libawutil.so.1.0.2
-rwxr-xr-x 1 root root 110500 Sep 29 2000 libawutil.so.1.0.0
-rwxr-xr-x 1 root root 110508 Sep 29 2000 libawutil.so.1.0.1
-rwxr-xr-x 1 root root 121760 Dec 24 2000 libawutil.so.1.0.2
...
```

Beim Linken von Programmen mit libawutil.so, libawutil.so.1, libawutil.so.1.0 oder libawutil.so.1.0.2 wird jetzt effektiv die Bibliothek libawutil.so.1.0.2 verwendet; so gelinkte Programme verwenden diese auch zur Laufzeit. Auch Programme. die früher bereits mit libawutil.so, libawutil.so.1, libawutil.so.1.0 gelinkt wurden, verwenden diese. Nur Programme, welche explizit mit libawutil.so.1.0.0 oder libawutil.so.1.0.1 gelinkt wurden, rufen die Funktionen daraus auf.

Wenn kurz danach eine erweiterte Version libawutil.so.1.1.0 installiert wird, die mehr Funktionalität bietet, aber rückwärtskompatibel zu libawutil.so.1.0.2 ist, dann hat das Verzeichnis so auszusehen:

```
klaus@aw33:~ > ls -l /usr/local/lib/
...
lrwxrwxrwx1 root root     14 Sep 29 2000 libawutil.so -> libawutil.so.1
lrwxrwxrwx1 root root     16 Dec 31 2000 libawutil.so.1 -> libawutil.so.1.1
lrwxrwxrwx1 root root     18 Dec 24 2000 libawutil.so.1.0 -> libawutil.so.1.0.2
lrwxrwxrwx1 root root     18 Dec 31 2000 libawutil.so.1.1 -> libawutil.so.1.1.0
-rwxr-xr-x1 root root 110500 Sep 29 2000 libawutil.so.1.0.0
-rwxr-xr-x1 root root 110508 Sep 29 2000 libawutil.so.1.0.1
-rwxr-xr-x1 root root 121760 Dec 24 2000 libawutil.so.1.0.2
-rwxr-xr-x1 root root 128106 Dec 31 2000 libawutil.so.1.1.0
...
```

Schade daß es unter Windows keine symbolischen Links gibt…

# A Anhang

## Literatur


[Systemzentrale] Andreas Jäger:
*Systemzentrale*
*Die C-Bibliothek in Linux-/Unix-Systemen*
c't 9/2001, Heise Verlag, Hannover
ISSN 0724-8679
Sehr lesenswert!

[CLib.Ref.] Sandra Loosemore, Roland McGrath, Andrew Oram, Richard M. Stallman:
*The GNU C Library Reference Manual*
Free Software Foundation http://www.gnu.org
Auf GNU-Systemen (beispielsweise Linux) wird von C-Programmen üblicherweise die *GNU C Library* verwendet. Diese enthält neben den ANSI-konformen noch eine wahnwitzige Sammlung von weiteren Funktionen, mit denen die jahrzehntelang auf Unix-Systemen gewachsene Funktionalität abgedeckt wird.
Dieses Buch ist die entsprechend umfangreiche Dokumentation der Bibliothek und üblicherweise in jeder Linux-Distribution enthalten.

[Linux Prog.] Goldt, v.d.Meer, Burkett, Wel:
*The Linux Programmer's Guide*
(Teil des *Linux Documentation Project*)

[KW: C] Klaus Wachtler:
*0 ist false, alles andere true*
*Eine Einführung in ANSI-C*
Selbstverlag, 2001
http://www.wachtler.de

[Stroustrup: C++] Bjarne Stroustrup:
*Die C++ Programmiersprache*
4., aktualisierte Auflage
Addison Wesley 2000
ISBN 3-8272-1660-X
Umfangreiche, vielleicht schwer verdauliche Einführung und Referenz; DAS Buch zu C++.

# Index

## Windows CEMENT