Konzepte von Betriebssystem-Komponenten

Programmstart & dynamische Bibliotheken

SS 05

Wladislaw Eckhardt

Wladi23@gmx.net

1 Einleitung

11. Problematik

12. Motivation

13. Lösungsansätze

14. Übersicht der Ausarbeitung

2 Bibliotheken

22. Definition

23. allgemein

24. statische Bindung

25. dynamische Bindung zum Programmstart

26. dynamische Bindung zur Laufzeit

3 Programmstart

3.1 Generierung eines Objektmoduls

32. Hauptspeichereinteilung fuer Prozesse

33. Programmstart mit Linken dynamischer Bibliotheken

4 Strategien der Verwendung dynamischer Bibliotheken

4.1 festes Linken

4.2 dynamisches Laden

5 Beispiele zur Erzeugung von Bibliotheken

5.1 statische Bibliothek

5.2 dynamische Bibliothek

6 Fazit

7 Literatur

1. 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.

1.1 Problematik

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.

1.2 Motivation

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.

1.3 Lösungsansatze

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*.

2. Bibliotheken

2.1 Definition

Bibliotheken im der Informationsverarbeitung sind Sammlung von

widerverwendbaren Funktionsmodulen, dadurch ist klare Abkapselung und Strukturierung von dem eigentlichen Programm möglich.

2.2 Bibliotheken allgemein

Es gibt sehr viele Arten von Bibliotheken, diese lassen sich nach verschiedenen Gesichtspunkten einteilen, z. B. nach Funktionsart. So gibt es Bibliotheken für grafische und mathematische Funktionen. Bibliotheken können auch nach der Programmiersprache eingeteilt werden: Java, Perl, C/C++.

Ich gehe hauptsächlich auf Systembibliotheken ein. 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 hinzu gebunden werden, hier gibt es drei Zeitpunktklassen: direkt nach der Programmübersetzung, bei jedem Programmstart und 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.

2.3 Statische Bindung

Das Binden von Programmen und Bibliotheken zur Entwicklungszeit (d.h bem Compilieren). 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 (engl. to link: verbinden). In Abb. 1 sieht man deutlich, dass jedes Programm mit den notwendigen Bibliotheken ein eigenständiges Modul ist.

Programme, die statisch mit Bibliotheken verbunden sind, nutzen einige der oben genannten 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.

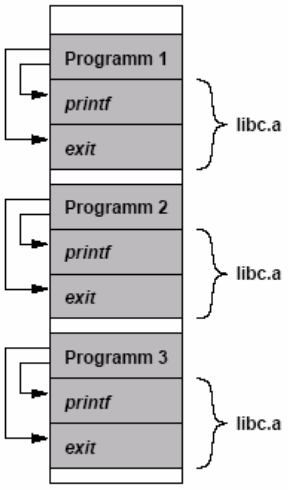

Abbildung 1: statische Bindung

2.4 dynamische Bindung zum Programmstart

Eleganter ist die Möglichkeit, Programme bei deren Start mit der passenden Bibliothek zu verbinden. Dies nennt man dynamisches Binden. Dies ist ein komplizierter 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, dieser Mechanismus ist besonders in der Abbildung 2 dargestellt, dabei benutzt jedes Programm die notwendigen Funktionen der Bibliothek in dem es diese referenziert. 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.

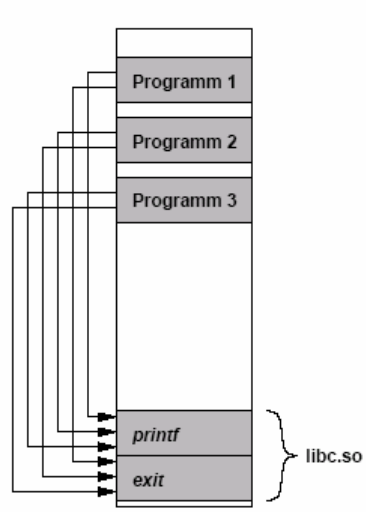

Abbildung 2: gemeinsam genutzte Bibliotheken

2.5 dynamische Bindung zur Laufzeit

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). In der Abbildung 3 wird gezeigt wie Bibliothek erst bei Bedarf in die Programme nachgeladen wird.

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.

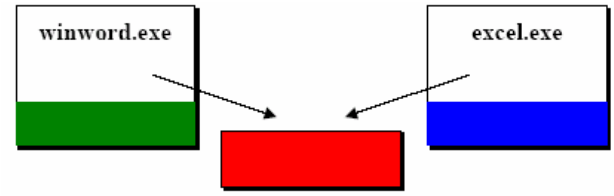

Abbildung 3: Bibliothek wird bei Bedarf in die Programme geladen

3. Programmstart

3.1 Generierung eines Objektmoduls

In Abbildung 4 wird gezeigt wie ein Modul generiert wird. Da wir bereits den Mechanismus der Erzeugung aus anderen Vorlesungen kennen gehe ich nur kurz darauf ein.

Der Programmerzeugungsprozess besteht aus mehreren Schritten:

- Der Präprozessor bearbeitet die Präprozessor-derektiven im Quelltext z. B. #include -oder #define Anweisungen
- Der Compiler erzeugt ein Objektmodul
- Der Linker erzeugt aus einem oder mehreren Objektmodulen ein lauffaehiges Programm. Die Objektmodule liegen entweder als einzelne Dateien (Suffix: *.o) oder in Form von Objektmodul-Bibliotheken (lib ...) vor.

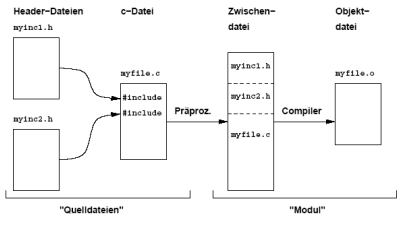

Abbildung 4: Generierung eines Objektes

3.2 Hauptspeichereinteilung für Prozesse

In *Abbildung 5 i*st eine abstrakte Darstellung der Speicherbereiche für ein Prozess dargestellt, das eigentliche Programm befinden sich im linken Teil der Abb., also im .code Segement, daneben ist das .data Segment für Statische Daten, .bss Segment fuer nicht initialisierte Dynamische Daten der genauso wie der Stack logisch unendlich gross ist physisch selbstverständlich begrenzt ist.

Abbildung 5: Einteilung des Hauptspeichers fuer Prozesse

3.3 Programmstart mit Linken dynamischer Bibliotheken

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.

4 Strategien der Verwendung dynamischer Bibliotheken

Auf beiden Systemen (Unix und Windows) hat man zwei Möglichkeiten, um

Funktionen (oder vollkommen analog: globale Variablen) einer dynamischen Bibliothek zu nutzen.

4.1 festes Linken

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. Funktionen.

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 Speicherbereiche eines Prozesses.

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.

Unter Windows entstehen beim Erzeugen der DLL zwei wichtige Dateien:

1. eine Bibliotheksdatei *irgendwas.lib*, welche die Dummyfunktionen zum Linken der Programme enthält, und
2. 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.

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.

4.2 dynamisches Laden

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*

Abbildung 6

5 Beispiele zur Erzeugung von Bibliotheken

5.1 statische Bibliotheken

Statische Bibliotheken erstellt man unter Unix mit dem Programm ***ar***. Es wird wie folgt aufgerufen:

ar <Schalter> <Libname> <Objektdateien>

Eine vollständige Liste der Schalter gibts in der *man* Dokumentation, zum Erstellen einer Bibliothek benutzt man i.A. "cr" für "Create" und "Replace", also eine Bibliothek erstellen, falls sie noch nicht existiert, und bereits vorhandene Objektdateien gleichen Namens ersetzen. Hier ein Beispiel:

ar cr libmylib.a modul1.o modul2.o modul3.o

Wie man hier bereits sieht, fangen Bibliotheksnamen immer mit "lib" an und hören mit ".a" auf. Unter Linux und einigen anderen Unixen gilt zudem, daß nur die *statischen* Bibliotheken die Endung **".a"** haben.

5.2 Dynamische Bibliotheken

Nicht alle Unix-Systeme beherrschen dynamisches Linken, und die Implementierungen können sich auch unterscheiden. Im Folgenden wird hier nur Linux betrachtet, das über eine moderne Implementierung dieses Konzeptes verfügt.

Um unter Linux dynamische Bibliotheken zu erstellen, müssen zunächst die Quelltextdateien mit dem zusätzlichen Schalter "-fPIC" zu Objektdateien übersetzt werden. Danach müssen sie mit weiteren speziellen Schaltern gelinkt werden. Das läßt sich am besten an einem Beispiel zeigen:

```
gcc -g -fPIC -c modul1.c
gcc -g modul1.o -shared -Wl,-soname,libirgendwas.so.1 -o libirgendwas.so.1.0
```

Wichtig sind hier u. a. die Compilerschalter in der Mitte (die Kommas gehören dazu!). Man beachte, daß der Name der Ausgabedatei eine Versionsnummer "hinter dem Komma" enthalten kann, aber der interne, in der Datei gespeicherte Bibliotheksname nur eine einstellige Versionsnummer hat.

Programme können je nach Notwendigkeit oder Belieben gegen eine bestimmte Version der Bibliothek (unter Angabe der einstelligen Versionsnummer) oder auch einfach nur gegen die Bibliothek gelinkt werden. In der Praxis wird die einstellige Version einer Bibliothek einfach als symbolischer Link auf die aktuelle zweistellige Version zur Verfügung gestellt.

Unter Linux wird der Datenbereich einer dynamischen Bibliothek für jeden sie verwendenden Prozeß individuell angelegt. Somit existiert zur Laufzeit der Programmcode einmal im Speicher, der Datenbereich jedoch unter Umständen vielfach.

6 Fazit

6.1 Vorteile dynamischer Bibliotheken gegenüber statischen Bibliotheken:

- Der ausführbare Code einer dynamischen Bibliothek wird von System nur einmal in den Speicher geladen, so dass alle Prozesse, die diese dynamische Bibliothek benutzen, den gleichen Code benutzen. Deswegen sollte man Code, der von mehreren Programmen benutzt werden kann, in eine dynamische Bibliothek packe. Das spart viel Speicher beim gleichzeitigen Ablauf dieser Programme.

- Diese Einsparung am Speicher bringt natürlich auch Geschwindigkeitsvorteile mit sich, da dadurch weniger Paging (Ein- und Auslagern von Speicherseiten) stattfinden.

- Da der Code einer dynamischen Bibliothek beim Linken nicht in das entsprechende Programm eingefügt wird, sind die aus dem Linken resultierenden Programme kleiner als solche, die mit statischen Bibliotheken gelinkt werden. Das spart zum einen Speicherplatz auf der Festplatte, zum anderen führt es auch zu *schnelleren Programmen*, da das Laden von

kleineren Programmen in den Arbeitsspeicher natürlich auch weniger Zeit beansprucht.

- Werden Fehler in einer dynamischen Bibliothek behoben oder eben nur erforderliche Änderungen (wie z. B. Code-Optimierungen) an ihr vorgenommen, so erfordert dies keine neue Generierung der Programme, die diese Bibliothek benutzen. Bei statischen Bibliotheken dagegen müsste man alle Programme, die diese Bibliothek nutzen, neu kompilieren und linken.

6.2 Nachteile dynamischer Bibliotheken gegenüber statischen:

- Ein mit dynamischen Bibliotheken gelinktes Programm ist für sich allein nicht ablauffähig, da es immer die zugehörigen dynamischen Bibliotheken benötigt. Das bedeutet, dass bei der *Auslieferung* dieses Programms niemals vergessen werden darf, die zugehörigen dynamischen Bibliotheken mitzuliefern, da das Programm sonst nicht läuft. Es sei denn, es handelt sich um dynamische Bibliotheken, die allgemein vom jeweiligen System angeboten werden.

- Da die von einem Programm benutzten dynamischen Bibliotheken beim Programmstart erst gesucht und geladen werden müssen, kann dies beim ersten Laden einer solchen Bibliothek zu einem zusätzlichen Zeitaufwand führen, den statische Bibliotheken nicht benötigen. Dieser *Nachteil* gilt jedoch nur für das erstmalige Laden! Da jedoch meist die entsprechenden dynamischen Bibliotheken schon für einen anderen Prozess in den Hauptspeicher geladen wurden, trifft diese auf den Grossteil von Programmen nicht mehr zu.

7 Literatur

Systemzentrale
Andreas Jäger:
Systemzentrale
Die C-Bibliothek in Linux-/Unix-Systemen, 2001

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

Stroustrup: C++
Bjarne Stroustrup:
Die C++ Programmiersprache
4., aktualisierte Auflage
Addison Wesley 2000

Moderne BS
2., überarbeitete Auflage 2002 von Andrew S. Tannenbaum

Referenz zu Abbildung 1, 4, 3:
http://www.wachtler.de/dynamischeBibliotheken/

Referenz zu Abbildung 2, 5:

http://www.cpp-entwicklung.de/cpplinux2/cpp_main/node6.html#SECTION006500000000000

http://www.willemer.de/informatik/cpp/dynlib.htm