Technische Universität Chemnitz-Zwickau
Fakultät für Informatik
Professur Betriebssysteme

# Diplomarbeit

## Dynamisches typsicheres Linken von C++-Programmen

eingereicht von: Andreas Eulitz
geb. am 21.7.1971 in Leisnig (Sa.)

Betreuer: Prof. Dr. Winfried Kalfa
Dipl.-Ing. Frank Schubert
Dipl.-Inf. Lutz Wohlrab

Chemnitz, den 30. August 1996

# Aufgabenstellung für die Diplomarbeit

## *Dynamisches typsicheres Linken von C++–Programmen*

---

Name: Andreas Eulitz
Matr.-Nr.: 7782

In komplexen Softwaresystemen gewinnt die Möglichkeit der dynamischen Adaption dieser Systeme zunehmend an Bedeutung. Aufgrund spezieller Anforderungen an diese Systeme ist es oft nicht möglich, diese Systeme anzuhalten und neu zu starten, um Korrekturen oder Erweiterungen am System vorzunehmen. In diesem Zusammenhang nimmt das dynamische Linken von Programmen, d. h. das Hinzufügen und Entfernen von Programmkomponenten während der Abarbeitung, eine zentrale Position ein. Hinzu kommen Versionsverwaltungsprobleme für die Komponenten in einem System, die geteilt von verschiedenen Applikationen genutzt werden (beispielsweise Shared-Libraries). Soll eine neue Version eines solchen Systems installiert werden, so existieren i. allg. noch Applikationen, welche die alte Version auch weiterhin benutzen möchten während andere, neu zu startende Applikationen, bereits die neue, ggf. erweiterte Version benutzen wollen. Außerdem haben sich in der Vergangenheit zunehmend objektorientierte Softwareentwicklungsmethoden und mit ihnen Standards für objektorientierte Programmiersprachen durchgesetzt (z.B. C++). Die geforderte strenge Typsicherheit dieser Sprachen bedingt besondere Mechanismen, um den Vorgang des Linkens ebenfalls typsicher zu gestalten.

Im Rahmen dieser Diplomarbeit soll daher eine Reihe von Problemen aus dem genannten Umfeld untersucht werden, wobei folgende inhaltliche Schwerpunkte zu beachten sind:

**Dynamisches Linken**

> Das dynamische Linken, Relinken und Unlinken von Programmen ist der Basismechanismus, der dynamischen Änderungen von Programmcode zugrundeliegt. Dabei ergibt sich eine Reihe von Besonderheiten aus der Benutzung von C++. Insbesondere Dienste, die sonst die Laufzeitumgebung erbringt (z. B. korrekter Aufruf der Konstruktoren global definierter Objekte), müssen jetzt vom dynamischen Linker realisiert werden.

**Typsicheres Linken von C++–Programmen**

> Um die von Sprachen wie C++ geforderte strenge Typsicherheit auch tatsächlich zu erreichen, muß der Vorgang des dynamischen Linkens ebenfalls typsicher gestaltet werden. Dafür sind geeignete Mechanismen erforderlich (spezielle Symbolnamen, alternative Objektfileformate).

**Versionsverwaltung**

> Die Probleme im Zusammenhang mit geteilt benutztem Code wie Shared-Libraries sind ein weiterer Schwerpunkt dieser Arbeit. Die Besonderheiten, die aus dem dynamischen Austausch einzelner Komponenten dieser Codebereiche resultieren, die entstehenden Versionskonflikte, allgemeine Ansätze für Linkvorgänge zwischen verschiedenen Adreßräumen und damit verbundene Schutzprobleme sind hierbei zu betrachten.

Grundlage für die Arbeit bildet die Untersuchung des aktuellen Standes der Forschung auf den genannten Gebieten. Vorhandene Lösungsansätze sind zu diskutieren und zu vergleichen. Ausgehend von diesen Untersuchungen sind Lösungsvorschläge für die einzelnen Probleme zu erarbeiten. Darauf aufbauend ist ein Prototyp für einen dynamischen Linker von C++–Applikationen in UNIX zu entwerfen und zu implementieren. Der Entwurf soll dabei die spätere Migration des Systems in eine Stand-Alone-Umgebung vorsehen. Zu diesem Zweck ist besonderes Augenmerk darauf zu richten, welche Dienste der Betriebssystemumgebung für die Arbeit des Linkers genutzt werden müssen.

Betreuer:
Prof. Dr. W. Kalfa,
Dipl.-Ing. F. Schubert,
Dipl.-Inf. L. Wohlrab
email: {kal,fsc,lwo}@informatik.tu-chemnitz.de

Technische Universität Chemnitz-Zwickau
Fakultät für Informatik
Professur Betriebssysteme

Dynamisches typsicheres Linken von C++-Programmen /
Eulitz, Andreas. - 1996. - 111 S., 39 Abb., 13 Tab., 23 Lit.
Chemnitz: Technische Universität Chemnitz-Zwickau, Fakultät für Informatik, Diplomarbeit

## Kurzreferat

Die vorliegende Arbeit untersucht Anforderungen, Probleme und Lösungsmöglichkeiten im Zusammenhang mit dem dynamischen Linken von C++-Programmen. Dabei wird speziell auf Aspekte der Typsicherheit, der Versionsverwaltung und der geteilten Benutzung von Objektmodulen eingegangen. Neben einer allgemeinen Einführung in Grundbegriffe und Methoden des Linkens erfolgt am Beispiel des Objektdateiformats ELF eine Erläuterung von speziellen Datenstrukturen und Verfahren in Verbindung mit dem dynamischen Linken. Abschließend wird die Implementation eines dynamischen Linkers vorgestellt.

Hinweis:
Bezeichnungen von Erzeugnissen, die zugleich eingetragene Warenzeichen sind, wurden nicht als solche kenntlich gemacht. Aus dem Fehlen der Markierung ® bzw. ™ kann nicht geschlossen werden, daß die Bezeichnung ein freier Warenname ist. Ebensowenig wird auf Patente oder Gebrauchsmusterschutz hingewiesen.

# Inhaltsverzeichnis

**Abkürzungsverzeichnis** **ix**

**1 Einleitung** **1**
1.1 Aufgabenstellung . . . 2
1.2 Vorgehensweise . . . 3

**2 Gegenstand und Methoden des Linkens** **5**
2.1 Linken während der Programmentwicklung . . . 5
2.2 Linken während der Programmausführung . . . 6
2.3 Objektmodule . . . 7
2.4 Symbole . . . 8
2.4.1 Namen . . . 8
2.4.2 Werte . . . 9
2.4.3 Typen . . . 10
2.4.4 Bindungsarten . . . 10
2.5 Relokationen . . . 11
2.5.1 Symbolische Relokation . . . 12
2.5.2 Nichtsymbolische Relokation . . . 12
2.6 Resolution . . . 13

**3 Konzepte des dynamischen Linkens** **15**
3.1 Explizites dynamisches Linken . . . 15
3.2 Implizites dynamisches Linken . . . 16
3.3 Abhängigkeiten von Objektmodulen . . . 16
3.4 Geteilte Benutzung . . . 17
3.5 Typsicherheit . . . 20
3.6 Versionsverwaltung . . . 21
3.7 Initialisierung und Terminierung globaler Datenobjekte . . . 21
3.8 Verzögertes Binden . . . 22
3.9 Linkgranularität . . . 23

**4 Das ELF-Objektdateiformat** **25**
4.1 Aufbau einer ELF-Datei . . . 25
4.2 Header . . . 26
4.3 Sektionen . . . 27
4.4 Segmente . . . 28

4.4.1 Textsegment . . . 29
4.4.2 Datensegment . . . 30
4.4.3 Dynamic-Segment . . . 30
4.5 Stringtabellen . . . 31
4.6 Symboltabellen . . . 32
4.7 Hashtabelle . . . 32
4.8 Relokationstabellen . . . 33
4.9 Bewertung von ELF . . . 34

**5 Dynamisches Linken auf der Basis von ELF 37**
5.1 Link-Editor . . . 37
5.2 Laufzeit-Linker . . . 39
5.3 Laden einer ELF-Objektdatei . . . 41
5.4 Abhängigkeitsgraph . . . 43
5.5 Initialisierung und Terminierung globaler Objekte . . . 43
5.6 Geteilte Benutzbarkeit . . . 44
5.7 Verzögertes Binden . . . 45
5.8 Positionsunabhängiger Code . . . 45
5.8.1 Global Offset Table . . . 46
5.8.2 Procedure Linkage Table . . . 49
5.9 Versionsverwaltung . . . 51

**6 Implementation eines dynamischen Linkers 53**
6.1 Vorbetrachtungen . . . 53
6.1.1 Globale Symbolmenge versus Gültigkeitsbereiche . . . 53
6.1.2 Abhängigkeitsgraph . . . 54
6.1.3 Funktionalität . . . 55
6.1.4 Relinken von Objektmodulen . . . 56
6.1.5 Re-Relokation . . . 58
6.1.6 Versionsverwaltung . . . 58
6.1.7 Typsicherheit . . . 59
6.1.8 Symbole der Linker-Applikation . . . 60
6.2 Nutzerschnittstelle . . . 61
6.3 Systemschnittstelle . . . 63
6.4 Datenstrukturen . . . 63
6.4.1 Klasse Module . . . 63
6.4.2 Klasse File . . . 64
6.4.3 Klasse Filelist . . . 65
6.4.4 Symboltabelle der Linker-Applikation . . . 65
6.5 Algorithmen . . . 65
6.5.1 Breite-zuerst-Durchmusterung eines Abhängigkeitsgraphen . . . 66
6.5.2 Tiefe-zuerst-Durchmusterung eines Abhängigkeitsgraphen . . . 67
6.5.3 Erzeugung eines Abhängigkeitsgraphen . . . 67
6.5.4 Löschen eines Abhängigkeitsgraphen . . . 68
6.5.5 Relokation . . . 68
6.5.6 Symbolsuche . . . 69
6.5.7 Relinken . . . 70
6.6 Probleme und Verbesserungsvorschläge . . . 70

**7 Zusammenfassung und Ausblick** **75**

**Literaturverzeichnis** **77**

**Glossar** **79**

**Abbildungsverzeichnis** **83**

**Tabellenverzeichnis** **85**

**Verzeichnis der Code-Beispiele** **87**

**Verzeichnis der Algorithmen** **89**

**A Implementationsbeschreibung** **91**
A.1 Verzeichnis- und Dateistruktur . . . . . 91
A.2 Parameter der Nutzerschnittstelle . . . . . 92
A.3 Systemschnittstelle des dynamischen Linkers . . . . . 94
A.4 Hinweise für die Kernel-Migration . . . . . 95
A.5 Anwendungsbeispiele . . . . . 97

**B Initialisierungs- und Terminierungscode** **103**
B.1 ITC mit dem Sun-C++-Compiler . . . . . 103
B.2 ITC mit dem GNU-C++-Compiler . . . . . 106

**Thesen** **109**

**Selbständigkeitserklärung** **111**

# Abkürzungsverzeichnis

ABI ..... Application Binary Interface
BFT ..... Breadth First Traversal
COFF ..... Common Object File Format
DFT ..... Depth First Traversal
ELF ..... Executable and Linking Format
GOT ..... Global Offset Table
ITC ..... Initialization and Termination Code
PHT ..... Programm Header Table
PIC ..... Position Independent Code
PLT ..... Procedure Linkage Table
SHT ..... Section Header Table
SVR4 ..... System V Release 4
TR ..... Technical Report

# Kapitel 1

# Einleitung

Die modulare Programmierung ist zum unverzichtbaren Bestandteil bei der Entwicklung komplexer Software-Systeme geworden. Die meisten Entwicklungsumgebungen unterstützen dieses Paradigma, indem sie die Aufteilung des Quelltextes eines Programms auf mehrere Dateien erlauben, die separat in Objektmodule übersetzt werden. Traditionellerweise verbindet ein Link-Editor diese Objektmodule während der Programmerzeugung zu einer ausführbaren Datei. Diese Vorgehensweise wird als *statisches Linken* bezeichnet. Werden Objektmodule erst zur Laufzeit zu einer ausführbaren Einheit verbunden, so spricht man von *dynamischen Linken*. Hierdurch wird die Modularisierung von der Ebene der Programmentwicklung auf die Ebene der Programmausführung erweitert.

Das Konzept des dynamischen Linkens hat eine lange Geschichte (s. [Kempf92]). Zu den ersten Betriebssystemen, in denen es eingesetzt wurde, zählen Multics und TENIX. In diesen Systemen wurde das dynamische Linken durch Kernel-Funktionen realisiert und während des Ladens von Programmen angewendet. Aufgrund des komplizierteren Ladevorgangs und der verlängerten Startzeit von Programmen wurde in Nachfolge-Betriebssystemen von diesem Konzept wieder abgesehen. Mit zunehmender Größe der zur Programmentwicklung eingesetzten Standard-Bibliotheken (und der damit zunehmenden Größe von Programmen) entstand ein erneutes Interesse am Konzept des dynamischen Linkens. In UNIX System V wurde es erstmals wieder aufgegriffen und als Bestandteil des Ladevorgangs von Programmen realisiert. Unter SunOS 4.x wurde der Laufzeit-Linker, das Werkzeug zur Durchführung des dynamischen Linkens, als separates Modul implementiert. Dieses Modul wird als Bestandteil von Anwendungsprogrammen im Rahmen eines Nutzerprozesses zur Ausführung gebracht. Über eine Programmierschnittstelle kann dabei auf die Funktionen des Laufzeit-Linkers zugegriffen werden. Betriebssysteme, die zu UNIX System V Release 4 konform sind (z. B. Solaris 2.x oder Linux), bedienen sich – ebenso wie BSD 4.4 – derselben Strategie. In OSF/1 wird das dynamische Linken durch den Programmlader realisiert. Eine Besonderheit hierbei ist, daß über verschiedene Adreßräume hinweg dynamisch gelinkt werden kann. Der OSF/1-Programmlader ist durch sogenannte Format-Handler in der Lage, Objektdateien verschiedener Formate zu verarbeiten. Ähnlich wie bei SunOS 4.x werden Funktionen für das dynamische Linken über eine Programmierschnittstelle verfügbar gemacht. Mit dem Software-Paket *Dld* (s. [Ho90]) wurde eine Routinen-Sammlung bereitgestellt, durch die eine Anwendung zur Laufzeit Programmkomponenten nach Bedarf laden und wieder aus dem Speicher entfernen kann.

Der Einsatz des dynamischen Linkens ermöglicht die Realisierung einer Reihe neuer Ziele, von denen beispielgebend die folgenden aufgezählt werden sollen:

- **dynamische Adaptierbarkeit von Software-Systemen**

  Die zunehmende Vielgestaltigkeit von Hard- und Software sowie der effiziente und dedizierte Einsatz spezieller Hardware macht die Adaptierbarkeit zu einer Schlüsseleigenschaft bei der Entwicklung von Software-Systemen (s. [Kalfa96]).

  Durch das dynamische Linken wird es möglich, unter Wahrung einer einheitlichen Schnittstelle die Funktionalität dieser Systeme zur Laufzeit an variable Bedingungen und Aufgabenbereiche anzupassen oder zu erweitern. Beispiele hierfür sind Netzwerkanwendungen, die durch das

dynamische Linken spezieller Programmkomponenten an die Verwendung verschiedener Kommunikationsprotokolle angepaßt werden, oder Datenbankanwendungen, die auf gleiche Weise die Zusammenarbeit mit unterschiedlichen Datenbankservern organisieren.

- **Software-Evolution durch den Austausch von Programmkomponenten**
  Veränderungen an traditionell erstellten Programmen werden erst nach deren erneuter Erzeugung wirksam. Dazu muß der Anwender neben dem Quelltext über die entsprechenden Entwicklungswerkzeuge (und Erfahrungen im Umgang mit diesen) verfügen. Durch das dynamische Linken wird die Distribution von erweiterten oder verbesserten Programmkomponenten ermöglicht, die direkt von einer Applikation verwendet werden können.

- **inkrementelle Programmentwicklung**
  Bei der herkömmlichen Programmentwicklung sind – unabhängig vom Arbeitsstand des Entwicklers – alle referenzierten Datenobjekte und Funktionen zu definieren. Aus diesem Grund werden Programmkomponenten anfänglich oft als Prototypen realisiert, die erst nach und nach mit der gewünschten Funktionalität ausgestattet werden. Das dynamische Linken erlaubt undefinierte Programmkomponenten unter der Voraussetzung, daß die Programmteile, die sie referenzieren, nicht durchlaufen werden. So kann sich der Entwickler auf die Erstellung und den Test einzelner Abschnitte einer Anwendung konzentrieren, während er andere Teile erst zu einem späteren Zeitpunkt implementiert.

- **Speicherplatzeinsparung**
  Bei der traditionellen Erstellung von Anwendungen unter Benutzung von Standardobjektmodulen wird deren Inhalt in die zu erstellende Datei kopiert. Bauen mehrere Anwendungen innerhalb eines Dateisystems auf dasselbe Objektmodul auf, existieren damit mehrere, völlig identische Kopien des Inhalts dieses Objektmoduls. In der Distribution einer Anwendung sind durch diese Vorgehensweise möglicherweise Teile enthalten, die als Standardobjektmodul auf dem Zielsystem schon existieren.

  Beim dynamischen Linken hingegen werden Objektmodule nicht kopiert, sondern referenziert. Sie sind also logischer, nicht aber physischer Bestandteil einer Anwendung. Damit braucht der Inhalt eines Standardobjektmoduls innerhalb eines Dateisystems nur einmal zu existieren. Bei der Auslieferung von Software kann auf den Inhalt von Standardobjektmodulen, die auf den Zielsystemen vorhanden sind, verzichtet werden.

  Das dynamische Linken ermöglicht darüber hinaus auch eine Verringerung des Hauptspeicherbedarfs. Dies wird erreicht, indem dieselbe Hauptspeicherrepräsentation eines Objektmoduls von mehreren Prozessen gleichzeitig benutzt werden kann.

- **Erhöhung der Portabilität**
  Durch das dynamische Linken wird eine Anwendung von Diensten getrennt, die durch Standardobjektmodule bereitgestellt werden. Wird diese Anwendung in einer veränderten Umgebung ausgeführt, wird sie dynamisch an die in dieser Umgebung gültigen Standardobjektmodule gebunden.

## 1.1 Aufgabenstellung

Das Ziel dieser Arbeit bestand in einer Analyse der Erfordernisse, Konzepte und Realisierungsmöglichkeiten des dynamischen Linkens von Programmkomponenten, die mit Hilfe der Programmiersprache C++ erstellt wurden. Dabei sollten Methoden für das Linken, Relinken und Unlinken dieser Komponenten als Basismechanismus des Verfahrens untersucht werden. Es galt, spezielle Aspekte, die sich aus der Verwendung von C++ ergeben, zu berücksichtigen. Dazu zählen insbesondere

- die Wahrung der Typsicherheit und
- die Ausführung von Initialisierungs- und Terminierungscode für die Abarbeitung der Konstruktoren und Destruktoren globaler Objekte.

Darüber hinaus sollten Maßnahmen, die die geteilte Benutzung von Programmkomponenten erfordern – so z. B. eine Versionsverwaltung – untersucht werden.

Aufbauend auf diesen Untersuchungen war ein Prototyp für ein Werkzeug zum dynamischen Linken zu implementieren. Zur Gewährung der Verständlichkeit und Wahrung von Portabilität und Flexibilität sollte die Implementation des Prototyps im Rahmen eines Anwendungsprogramms auf der Basis des Betriebssystems UNIX erfolgen. Um eine Integration der Ergebnisse der vorliegenden Arbeit in laufende Forschungen (s. [Kalfa96]) zu erleichtern, wurde eine Realisierung speziell auf dem UNIX-Derivat Linux angestrebt.

Aus den in [Schubert96] dargestellten Gründen sollte zur Entwicklung des dynamischen Linkers die Programmiersprache C++ Verwendung finden. Im Sinne der Aufgabenstellung galt es, mittels derselben Programmiersprache erstellte Objektdateien als Eingabe des dynamischen Linkers zu verarbeiten. Zur Generierung dieser Objektdateien sollten die auf der Entwicklungsplattform verfügbaren Entwicklungswerkzeuge in möglichst unveränderter Form zum Einsatz kommen.

Bei der Implementation war eine spätere Kernel-Migration zu berücksichtigen. Dies machte eine Dokumentation der Systemschnittstelle notwendig, und erforderte den weitestgehenden Verzicht auf Mechanismen, die sich nicht mit existierenden oder angestrebten Systemdiensten verwirklichen ließen.

## 1.2 Vorgehensweise

Die vorliegende Arbeit beginnt mit einer Einordnung des Linkens in den Prozeß von Programmerzeugung und -ausführung. Nach einer Darstellung der grundlegenden Begriffe und Verfahren werden spezielle Konzepte im Zusammenhang mit dem dynamischen Linken erläutert. Anschließend erfolgt eine kurze Beschreibung des ELF-Objektdateiformats. Hierbei wird besonderes Augenmerk auf die Elemente des Formats gelegt, die für das dynamische Linken von Bedeutung sind. Danach wird gezeigt, wie die Konzepte des dynamischen Linkens basierend auf dem ELF-Objektdateiformat realisiert werden. Nach einer ausführlichen Vorbetrachtung zu den Anforderungen und Entwurfsaspekten wird die Implementation eines Werkzeugs zum dynamischen Linken vorgestellt. Abschließend erfolgt die Diskussion von Verbesserungsvorschlägen und Erweiterungen der Implementation. Der Anhang geht detaillierter auf Schnittstellen und Anwendungsbeispiele des dynamischen Linkers ein.

Die im Rahmen dieser Arbeit vorgenommenen Untersuchungen wurden auf den folgenden Plattformen durchgeführt:

1. SPARC unter Solaris 2.x
2. Intel i386 unter Linux 1.7.x

Programmentwicklung und -test erfolgte auf diesen Plattformen mittels des Sun-C++-Compilers (1.) bzw. des GNU-C++-Compilers (1. und 2.). Programmiersprachenspezifischen Ausführungen und Quelltextbeispielen liegt C bzw. C++ zugrunde. Kommandozeilenbeispiele werden in Bourne-Shell-Syntax angegeben. Wird im Rahmen dieser Arbeit von Entwicklungssystemen gesprochen, so wird vereinfachend davon ausgegangen, daß es sich hierbei um auf Compilern basierende Systeme handelt. Zur Erklärung der verwendeten Begriffe sei auf das Glossar am Ende dieser Arbeit verwiesen.

# Kapitel 2

# Gegenstand und Methoden des Linkens

Dieses Kapitel stellt den Link-Editor und den Laufzeit-Linker als die grundlegenden Werkzeuge des Linkens vor. Es werden die Datenstrukturen, über denen beide Programme operieren, erläutert. Dazu zählen die als Eingabe verarbeiteten Dateien ebenso wie die darin zur Auflösung symbolischer Referenzen enthaltenen Informationen, die zusammen mit den darauf beruhenden Verfahren näher erklärt werden.

## 2.1 Linken während der Programmentwicklung

Während der Programmentwicklung erfolgt eine stufenweise Überführung einer Repräsentationsform eines Programms in eine andere. Der Quelltext einer Programmiersprache wie C oder C++ wird dabei meist von einem Compiler über mögliche Zwischenformen (z. B. Assembler-Quelltext) in Maschinensprache übersetzt und in Objektmodulen gespeichert. Diese werden vom Link-Editor z. T. unter Zuhilfenahme von Standardobjektmodulen zu neuen Objektmodulen oder ausführbaren Dateien verknüpft. Abbildung 2.1 stellt diesen Prozeß dar, bei dem sich von Ebene zu Ebene die Anzahl der erzeugten Einzeloperation erhöht, während gleichzeitig deren Komplexität und Abstraktionsniveau abnimmt.

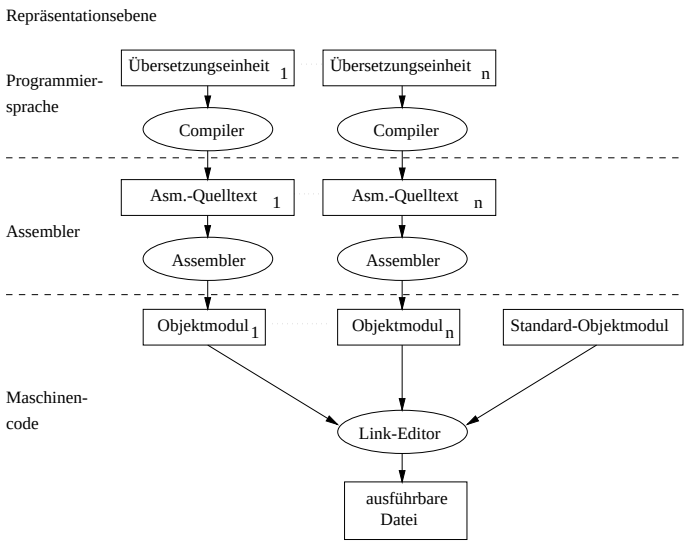


Abbildung 2.1: Einsatz des Linkers bei der Programmentwicklung

Maschinencode kann von der Hardware eines Rechners direkt interpretiert werden und bedarf daher keiner weiteren Übersetzung. Jedoch muß der Maschinencode innerhalb eines Objektmoduls in einer Form angeordnet und mit zusätzlichen Informationen versehen werden, die dem Link-Editor einen Zugriff auf die logischen Einheiten eines Objektmoduls ermöglichen. Objektmodule müssen darüber hinaus Informationen enthalten, die es dem Betriebssystem ermöglichen, den Inhalt dieser Module in den Adreßraum eines Prozesses zu laden, an die Position innerhalb des Adreßraums anzupassen und möglicherweise zur Ausführung zu bringen.

Dies bedingt, daß Objektmodule einer genau definierten Struktur unterliegen müssen. Diese Struktur wird als *Objektdateiformat* bezeichnet. Werkzeuge zur Programmentwicklung und Komponenten des Betriebssystems müssen auf das Objektdateiformat abgestimmt werden.

Durch das Objektdateiformat wird der gezielte Zugriff auf die Maschinensprachedarstellung von Strukturen der Programmiersprache (z. B. Datenobjekte oder Funktionen) geregelt. Zu diesem Zweck werden symbolische Informationen über diese Strukturen verwaltet, die eine Identifikation des entsprechenden Speicherabschnitts durch einen Namen ermöglichen. Nutzungsbeziehungen zwischen Objektmodulen können somit abstrakt durch Symbolnamen dargestellt werden.

Die Aufgabe des Link-Editors besteht darin, Objektmodule zu einer Einheit zu verbinden. Traditionellerweise werden dabei die Inhalte der Objektmodule zu einer Ausgabedatei konkateniert. Bei der Erstellung einer ausführbaren Datei löst der Link-Editor anschließend die abstrakten Nutzungsbeziehungen auf, indem er Referenzen zu Symbolnamen durch die konkrete Position des durch den Symbolnamen bezeichneten Speicherabschnitts ersetzt. Diese Vorgehensweise wird als *statisches Linken* bezeichnet.

Mit dem *dynamischen Linken* verändert sich die Aufgabe des Link-Editors. Objektmodule werden hierbei nicht in die zu erstellende Datei kopiert, sondern von dieser referenziert. Damit ist die Konkatenationsoperation durch die Erzeugung von Referenzen zu Objektmodulen zu ersetzen. Die endgültige Position des Inhalts eines Objektmoduls innerhalb des Adreßraums eines Prozesses ist damit nicht mehr vorherbestimmt und symbolische Referenzen können erst während der Initialisierung eines Prozesses aufgelöst werden.

## 2.2 Linken während der Programmausführung

Dynamisches Linken bedeutet, daß Objektmodule erst zur Laufzeit zu einer ausführbaren Einheit verbunden werden. Zu diesem Zweck müssen beim Start einer ausführbaren Datei die darin referenzierten Objektmodule zusätzlich zur ausführbaren Datei in den Speicher geladen werden. Anschließend sind symbolische Referenzen zwischen den Objektmodulen (die ausführbare Datei ist ebenfalls als Objektmodul zu verstehen) aufzulösen. Für diese Aufgabe ist eine dem Link-Editor in der Funktion ähnliche Komponente – der *Laufzeit-Linker* – notwendig.

Zur Implementation des Laufzeit-Linkers ist eine Erweiterung des Programmladers (z. B. `exec(2)`) vorstellbar. Dieser Weg wird jedoch innerhalb der untersuchten Betriebssysteme nicht beschritten. Der Laufzeit-Linker wird vielmehr selbst durch ein Objektmodul dargestellt, das logischer Bestandteil eines jeden dynamisch gelinkten Programms ist. Der Programmlader muß neben der ausführbaren Datei somit auch den Laufzeit-Linker laden und diesen statt des Inhalts der ausführbaren Datei starten. Nachdem der Laufzeit-Linker seine (initiale) Arbeit beendet hat, ruft er den Code der ausführbaren Datei auf.

Neben der Verwendung während des Programmstarts wird der Laufzeit-Linker auch zum dynamischen Linken von Objektmodulen auf *explizite Anforderung* hin eingesetzt. Eine solche Anforderung wird durch den Aufruf von Funktionen des Laufzeit-Linkers aus dem Code einer Anwendung heraus erzeugt. Wurde ein Objektmodul auf diese Weise gelinkt, muß der Zugang zu den darin enthaltenen Datenobjekten und Funktionen durch die Angabe von Symbolnamen ermöglicht werden. Diese Aspekte führen zur Definition einer *Programmierschnittstelle* des Laufzeit-Linkers. Die Verwendung der Programmierschnittstelle bewirkt, daß der Laufzeit-Linker auch nach dem Programmstart aktiv wird. Nutzungsbeziehungen zwischen Objektmodulen können damit in Abhängigkeit vom Programmlauf hergestellt werden.

## 2.3 Objektmodule

Objektmodule bilden die Eingabe von Link-Editor und Laufzeit-Linker. Als Objektmodul werden im Rahmen dieser Arbeit

- Objektdateien,
- Elemente von Objektdatei-Kollektionen und
- Hauptspeicherrepräsentationen von Objektdateien

bezeichnet.

Ein Beispiel für Objektdatei-Kollektionen bilden die mittels des `ar(1)`-Kommandos erzeugten *Archive*, denen durch das `ranlib(1)`-Kommando eine Archiv-Symboltabelle hinzugefügt wurde[1]. Die Archiv-Symboltabelle ist eine Zusammenfassung aller symbolischen Informationen der Elemente eines Archivs. Damit kann gezielt auf einzelne Objektdateien im Archiv zugegriffen werden. Dateinamen von Archiven werden meist mit dem Suffix `.a` versehen.

Objektdateien bestehen aus Maschinencode und Datenobjekten. Neben Informationen zur Auflösung symbolischer Referenzen enthalten Objektdateien zusätzlich eine Vielzahl „administrativer" Angaben, so z. B:

- das Rechnersystem, auf dem ihr Inhalt abgearbeitet werden kann
- Wortbreite und Codierungsart
- Informationen, wie ihr Inhalt in den Speicher zu laden ist

Der genaue Aufbau einer Objektdatei wird durch das verwendete Objektdateiformat bestimmt. Beispiele für Objektdateiformate sind ELF, COFF oder XCOFF.

Gemäß ihrer Verwendung soll im folgenden zwischen vier Arten von Objektmodulen unterschieden werden:

1. **relozierbare Objektdateien**
   Eine relozierbare Objektdatei (*relocatable object*) entsteht als Ergebnis des Übersetzungsprozesses oder durch die Konkatenation bereits existierender relozierbarer Objektdateien durch den Link-Editor. Im weiteren Verlauf werden relozierbare Objektdateien als Eingabe des Link-Editors bei der Erstellung geteilt benutzbarer Objektdateien sowie statisch oder dynamisch gelinkter Programme verwendet. Relozierbare Objektdateien sind meist am Suffix `.o` zu erkennen.
2. **statisch gelinkte Programme**
   Ein statisch gelinktes Programm (*static executable*) stellt eine abarbeitungsbereite Anwendung dar. Ein solches Programm entsteht mit der Konkatenation relozierbarer Objektdateien durch den Link-Editor. Eine dieser relozierbaren Objektdateien muß den Eintrittspunkt des Programms definieren (die `main()`-Funktion in C). Ein statisch gelinktes Programm unterliegt keinen weiteren Link-Schritten und darf daher keine unaufgelösten symbolischen Referenzen enthalten.
3. **geteilt benutzbare Objektdateien**
   Geteilt benutzbare Objektdateien (*shared objects*) werden vom Link-Editor zur Erstellung dynamisch gelinkter Programme verwendet und vom Laufzeit-Linker bei der Abarbeitung dieser Programme benutzt. Entscheidendes Merkmal geteilt benutzbarer Objektdateien ist, daß deren Hauptspeicherrepräsentation innerhalb mehrerer Prozesse verwendet werden kann. Eine geteilt benutzbare Objektdatei entsteht durch die Konkatenation relozierbarer Objektdateien durch den Link-Editor. Geteilt benutzbare Objektdateien können nicht aufgelöste symbolische Referenzen enthalten, die der Laufzeit-Linker mit den Symboldefinitionen anderer geteilt benutzbarer Objektdateien oder dynamisch gelinkter Programme verbindet. Geteilt benutzbare Objektdateien sind meist am Suffix `.so` zu erkennen.

[1] Auf einigen Systemen beinhaltet `ar(1)` die Funktionalität von `ranlib(1)`.

4. **dynamisch gelinkte Programme**

   Ein dynamisch gelinktes Programm (*dynamic executable*) stellt ähnlich wie ein statisch gelinktes Programm eine abarbeitungsbereite Anwendung dar. Im Gegensatz zu statisch gelinkten Programmen werden von dynamisch gelinkten Programmen jedoch geteilt benutzbare Objektdateien referenziert. Diese werden vom Laufzeit-Linker während des Programmstarts (oder danach) in den Adreßraum des entsprechenden Prozesses eingebracht. Durch den Laufzeit-Linker werden symbolische Referenzen zwischen den geteilt benutzbaren Objektdateien und dem dynamisch gelinkten Programm aufgelöst.

   Ein dynamisch gelinktes Programm entsteht durch die Verarbeitung relozierbarer und geteilt benutzbarer Objektdateien durch den Link-Editor. Für die relozierbaren Objektdateien gilt das gleiche wie im Fall statisch gelinkter Programme, geteilt benutzbare Objektdateien hingegen werden als sogenannte *Abhängigkeiten* im zu erstellenden Programm vermerkt.

## 2.4 Symbole

Durch den Übersetzungsprozeß werden Elemente der Programmiersprache wie globale Datenobjekte oder Funktionen in zusammenhängende Bereiche der Objektdatei abgebildet. Da die Position des Inhalts eines Objektmoduls innerhalb einer Anwendung (Datei oder Prozeß) nicht vorherbestimmt ist, können diese Elemente nicht durch eine Positionsangabe identifiziert werden. Hierzu ist eine abstrakte, positionsunabhängige Identifikation notwendig. Dafür bietet sich die Verwendung eines Namens an. Durch eine *Symboldefinition* wird einem Element der Objektdatei ein solcher Name zugeordnet.

So werden z. B. für die folgenden Elemente der Programmiersprachen C/C++ Symboldefinitionen erzeugt:

- auf Dateiebene definierte Datenobjekte
- Funktionen
- `static`-Variablen von Funktionen
- Klassenvariablen
- Zeichenketten-Literale

Symboldefinitionen stellen die Kombination aus einem Symbolnamen und einer Symbolbeschreibung dar. Die Symbolbeschreibung ist eine Menge von Attributen, die dazu dienen, das repräsentierte Element zu lokalisieren, seinen Typ zu spezifizieren und die Verwendbarkeit der Symboldefinition anzuzeigen.

Die einer Symboldefinition zugrundeliegende Datenstruktur wird auch für *undefinierte Symbole* benutzt. Durch undefinierte Symbole werden die Namen von Elementen vermerkt, die innerhalb eines Objektmoduls referenziert, nicht aber definiert werden (s. Abschn. 2.5.1, S. 12). Undefinierte Symbole werden in Anlehnung an höhere Programmiersprachen auch als *Symboldeklarationen* bezeichnet (s. [Gircys88]). Der Term *Symbol* wird kontextabhängig sowohl für definierte als auch für undefinierte Symbole gebraucht. Die Symbole eines Objektmoduls werden in einer *Symboltabelle* zusammengefaßt.

Die folgenden Abschnitte enthalten eine nähere Beschreibung von Symbolnamen sowie der Attribute Wert, Typ und Bindung der Symbolbeschreibung.

### 2.4.1 Namen

Zwischen verschiedenen Objektmodulen werden Symbole durch ihren Namen identifiziert. Innerhalb eines Objektmoduls können unbenannte Symbole Verwendung finden, die durch ihren Symboltabellen-Index identifiziert werden.

Auf den im Rahmen dieser Arbeit als Plattform verwendeten Systemen unterliegen Symbolnamen praktisch keiner Längenbeschränkung[2].

Im Fall der Programmiersprache C werden Symbolnamen erzeugt, die den Quelltext-Namen der entsprechenden Elemente gleichen (auf den untersuchten Plattformen werden dabei Quelltext-Namen nicht wie auf älteren Systemen um ein führendes Underscore-Zeichen ergänzt). Ausnahmen bilden dabei `static`-Variablen von Funktionen und Zeichenketten-Literale, die compilerabhängig zu Symbolen führen, die entweder keinen oder einen automatisch generierten Namen tragen.

Im Fall von C++ werden Symbolnamen in gleicher Weise wie für C generiert. Eine Ausnahme bilden hierbei jedoch Funktionen (Methoden) und Klassenvariablen, für die sogenannte *dekorierte Symbolnamen* (*mangled names*) erzeugt werden. Hierbei wird der Quelltext-Name einer Funktion um eine sogenannte *Signatur* ergänzt, die Informationen über Anzahl und Typ der Funktionsargumente beinhaltet. Durch diese Maßnahme können vom Linker polymorphe Funktionen unterschieden werden. Namen von Symbolen, die für Members einer Klasse erzeugt werden (Methoden, Klassenvariablen), enthalten darüber hinaus eine Codierung des Klassennamens. Für Operator-Funktionen werden spezielle Namenskürzel verwendet.

Durch das Dekorieren von Symbolnamen kann die Typsicherheit beim Linken von C++-Programmen durch die Übereinstimmung von Symbolnamen realisiert werden. Die Mechanismen für das Dekorieren von Symbolnamen sind nicht standardisiert und von Compiler zu Compiler verschieden (als Beispiel sei hier auf [Sun92] verwiesen). Damit können u. U. Objektmodule, die mit verschiedenen Compilern erstellt wurden, nicht gelinkt werden. Diese Vorgehensweise ist laut [Ellis91] bewußt gewählt worden, da das binäre Layout von Objekten der Sprache C++ nicht standardisiert ist. Auf diese Weise wird verhindert, daß Objektmodule miteinander verbunden werden können, die bezüglich des Objektlayouts inkompatibel sind.

### 2.4.2 Werte

Eine Symboldefinition identifiziert i. allg. die Maschinenspracherepräsentation eines Elements der Programmiersprache. Diese Maschinenspracherepräsentation soll im folgenden Text als *Symbolwert* bezeichnet werden. Der Symbolwert wird meist durch eine Positionsangabe identifiziert. Die Positionsangabe kann in Abhängigkeit von der Art des Objektmoduls eine relative Adresse oder eine Dateiposition sein. Darüber hinaus kann der Symbolwert zusätzlich durch eine Größenangabe beschrieben werden.

Für die Werte nichtinitialisierter Variablen wird innerhalb einer Objektdatei meist kein Speicherplatz reserviert. Vielmehr geben spezielle Symboldefinitionen die Position in einem Hauptspeicherabschnitt an, der erst zur Laufzeit allokiert wird. Dieser Hauptspeicherabschnitt wird typischerweise als `.bss`-Sektion oder -Segment bezeichnet und meist mit 0 initialisiert. Die so gebrauchten Symboldefinitionen werden als *vorläufige Symbole* (*tentative* oder *common symbols*) bezeichnet. Durch den Wert vorläufiger Symbole werden Parameter für die Speicherplatzreservierung – etwa Größe und Ausrichtungsanforderung – bereitgestellt. Vorläufige Symbole dienen der Speichereinsparung in Dateien. So führt z. B. die C-Definition

```
char c[1024];
```

zu einem vorläufigen Symbol, dessen Wert keinen Platz in der entsprechenden Objektdatei einnimmt, und der erst nach dem Programmstart 1 KB im Hauptspeicher belegt.

Desweiteren existieren Symbole, deren Wert nur durch Informationen des Symboltabelleneintrags verkörpert wird. Diese Symbole werden als *absolute Symbole* bezeichnet. So werden beispielsweise die Namen von Quelltextdateien, aus denen ein Objektmodul hervorgegangen ist, von den meisten Compilern in Form eines absoluten Symbols im Objektmodul vermerkt:

```
$ nm hello.o
```

[2]Bis zu 100 Zeichen lange Symbolnamen wurden problemlos verarbeitet.

```
Symbols from hello.o:

[Index]   Value      Size    Type  Bind  Other Shndx   Name

[1]     |         0|       0|FILE |LOCL |0    |ABS    |hello.c
...
```

Absolute Symbole stellen administrative Informationen für den Linkprozeß dar und sollten nicht aus einem Programm heraus referenziert werden.

### 2.4.3 Typen

Der *Symboltyp* bezeichnet den Typ des Symbolwerts. Hierbei wird mindestens zwischen den folgenden Symboltypen unterschieden (Objektdateiformate ELF und COFF):

- Datenobjekt
- Programmcode
- kein Typ

Im Fall einer Symboldefinition identifiziert der Symboltyp das repräsentierte Element der Programmiersprache als Programmcode (z. B. als Funktion) oder Datenobjekt (z. B. als Variable). Undefinierten Symbolen ist hingegen meist kein Symboltyp zugeordnet.

Es existieren Objektdateiformate, die eine detailliertere Beschreibung von Symboltypen zulassen. So können z. B. unter dem auf HP-UX eingesetzten a.out-Format mittels spezieller Deskriptoren die Typen von Datenobjekten sowie Rückgabewerten und Parametern von Funktionen beschrieben werden. Ein solcher Deskriptor legt u.a. den entsprechenden Grunddatentyp der Architektur (byte, word, ...) fest, identifiziert das bezeichnete Objekt als Aggregat (Feld, Struktur) und zeigt ggf. an, ob es sich um einen Wert- oder Referenzparameter handelt.

### 2.4.4 Bindungsarten

Die *Bindungsart* zeigt den Gültigkeitsbereich einer Symboldefinition an und regelt den Vorrang gleichnamiger Symboldefinitionen. Unter einigen Objektdateiformaten (z. B. COFF) wird die Bindungsart auch als *Speicherklasse* eines Symbols bezeichnet. Im allgemeinen wird zwischen den folgenden Bindungsarten unterschieden:

- globale Bindung
- lokale Bindung

Symboldefinitionen mit globaler Bindung können aus anderen Objektmodulen heraus referenziert werden; sie sind im Rahmen des Linkprozesses also „global“ gültig. Im Fall der Programmiersprache C kann durch eine Anweisung der Form

```
int i = 0;
```

auf Dateiebene eine global gebundene Definition für das Symbol `i` erreicht werden.

Im Gegensatz dazu lassen sich lokale Symbole nur innerhalb der Objektdatei referenzieren, in der sie definiert sind. Lokal gebundene Symbole lassen sich in C durch die Vergabe der Speicherklasse `static` für auf Dateiebene definierte Datenobjekte oder Funktionen erzeugen. So bewirkt also

```
static int i = 0;
```

eine lokal gebundene Definition für das Symbol `i`. Oftmals werden Symbole für Funktionen und Datenobjekte, die nicht über eine Schnittstelle exportiert werden sollen, lokal gebunden. Damit kann auf der Ebene der Objektmodule eine einfache Datenkapselung realisiert werden.

Bei der Auflösung symbolischer Referenzen eines Objektmoduls genießen die darin enthaltenen lokalen Symbole Vorrang vor gleichnamigen globalen Symbolen anderer Objektmodule.

Im Rahmen des Objektdateiformats ELF werden darüber hinaus auch sogenannte *schwache Bindungen* (*weak bindings*) unterschieden. Mittels dieser Bindungsart können Alias-Symbole für global oder lokal[3] gebundene Symboldefinitionen bereitgestellt werden. Dabei genießen gleichnamige lokal oder global gebundene Symboldefinitionen Vorrang vor schwach gebundenen. Zweck und Verwendung schwach gebundener Symbole soll an folgendem Beispiel illustriert werden:

Ein Objektmodul $A$ definiert eine generische Sortierfunktion `sort()` und erwartet hierfür, daß von einem nutzenden Objektmodul $B$ die Ordnungsrelation in Form einer Funktion `rel()` bereitgestellt wird. Das Modul $A$ selbst definiert eine Standardfunktion `_rel()` und richtet unter dem Namen `rel()` ein schwach gebundenes Symbol für diese Standardfunktion ein (s. Code-Beispiel 2.1).

```
...
int _rel(int x, int y){
  return( x < y ? 1 : 0);
}

#pragma weak rel = _rel

void sort(int f[], ...){
  ...
  if( rel( f[i], f[i + 1])) ...
  ...
}
...
```

**Code–Beispiel 2.1:** Verwendung schwach gebundener Symbole

Definiert $B$ eine Funktion `rel()`, so führt das zu einer global gebundenen Symboldefinition, die Vorrang vor dem gleichnamigen schwach gebundenen Symbol aus $A$ genießt. Anderfalls findet über das schwach gebundene Symbol die Standardfunktion `_rel()` aus $A$ Verwendung[4].

## 2.5 Relokationen

Zum Zeitpunkt der Erzeugung eines Objektmoduls steht i. allg. nicht fest, auf welche Position im Adreßraums eines Prozesses der Inhalt dieses Moduls später plaziert wird. Absolute Adressen innerhalb eines Objektmoduls müssen daher, nachdem die Position des Objektmoduls feststeht, neu bestimmt werden. Der Vorgang dieser Neubestimmung wird als *Relokation* bezeichnet.

Statisch gelinkte Programme werden vom Betriebssystem meist an konstante Adressen geladen. Damit steht die Position der Inhalte der beteiligten Objektmodule nach deren Konkatenation zur ausführbaren Datei fest. Relokationen können somit zum Zeitpunkt der Erstellung eines solchen Programms ausgeführt werden. Auf den untersuchten Systemen übernimmt der Link-Editor diese Aufgabe.

Im Fall dynamisch gelinkter Programme steht die Position des Inhalts von Objektmodulen im Adreßraum des benutzenden Prozesses erst nach dem Laden der Module durch den Laufzeit-Linker fest. Relokationen müssen daher vom Laufzeit-Linker ausgeführt werden.

[3] Durch die Bereitstellung eines Aliasnamens für ein lokales Symbol kann die Datenkapselung eines Objektmoduls unterlaufen und der Wert des lokalen Symbols von anderen Objektmodulen benutzt werden.

[4] Dieses Beispiel dient nur der Illustration, in der Praxis würde `sort()` mittels eines Parameters ein Zeiger auf `rel()` übergeben.

Der von einer Relokation modifizierte Speicherbereich (z. B. die Zieladresse eines Sprungbefehls) wird im folgenden als *Relokationsfeld* bezeichnet. Die Art, wie das Relokationsfeld zu modifizieren ist und welche Werte dafür Verwendung finden, wird durch den *Relokationstyp* bestimmt. Relokationsfeld und Relokationstyp sind Teil einer *Relokationsinformation*. Relokationsinformationen werden innerhalb eines Objektmoduls in *Relokationstabellen* aufbewahrt.

Wird das Relokationsfeld mittels eines Symbolwerts neu bestimmt, so spricht man von *symbolischen Relokationen*, sonst von *nichtsymbolischen Relokationen*. Die folgenden beiden Abschnitte beschreiben beide Relokationsarten näher.

### 2.5.1 Symbolische Relokation

Durch definierte und undefinierte Symbole können Nutzungsbeziehungen zwischen Objektmodulen mittels Symbolnamen auf abstrakte, positionsunabhängige Weise dargestellt werden. Zur Laufzeit müssen diese abstrakten Nutzungsbeziehungen in konkrete Adreßangaben überführt werden. Dies geschieht mittels *symbolischer Relokationen*, in deren Folge ein Relokationsfeld mit der Laufzeit-Adresse eines Symbolwerts (z. B. einer Variable oder einer Funktion) überschrieben wird.

Zu diesem Zweck verweist die zugehörige Relokationsinformation auf einen Symboltabelleneintrag innerhalb des Objektmoduls, der den Namen des benötigten Symbolwerts angibt. Hierbei handelt es sich um ein definiertes Symbol, wenn das Objektmodul den benötigten Symbolwert enthält, andernfalls um ein undefiniertes Symbol. Die Kombination aus einer solchen Relokationsinformation und dem entsprechenden Symboltabelleneintrag soll auch als *Symbolreferenz* bezeichnet werden.

Verweist eine Relokationsinformation auf ein undefiniertes Symbol, müssen andere Objektmodule nach der gleichnamigen Symboldefinition durchsucht werden (s. Abb. 2.2).

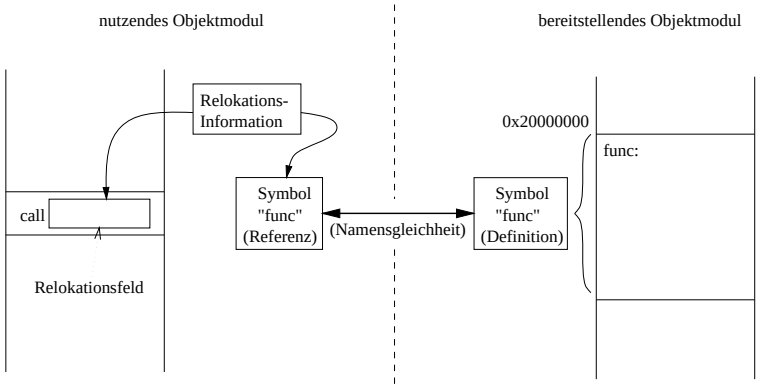


Abbildung 2.2: Situation vor Durchführung einer symbolischen Relokation

Bei der Suche nach einer Symboldefinition kann es vorkommen, daß mehrere gleichnamige Definitionen gefunden werden. In diesem Fall wird mittels sogenannter Resolutionsregeln (s. Abschn. 2.6, S. 13) entschieden, welche dieser Definitionen zu verwenden ist.

Wurde eine Symboldefinition gefunden, so wird das Relokationsfeld mit der Adresse des Symbolwerts überschrieben (s. Abb. 2.3).

### 2.5.2 Nichtsymbolische Relokation

Es existieren Relokationen, bei denen sich der Inhalt des Relokationsfeldes nicht aus der Adresse eines Symbolwerts bestimmt. Diese Relokationen sollen als *nichtsymbolische Relokationen* bezeichnet werden.

Ein Beispiel hierfür stellt die Relokation des Operanden eines absolut adressierten Sprungbefehls dar. So kann bei der Erzeugung des Objektmoduls das Sprungziel bezüglich einer Basisadresse 0

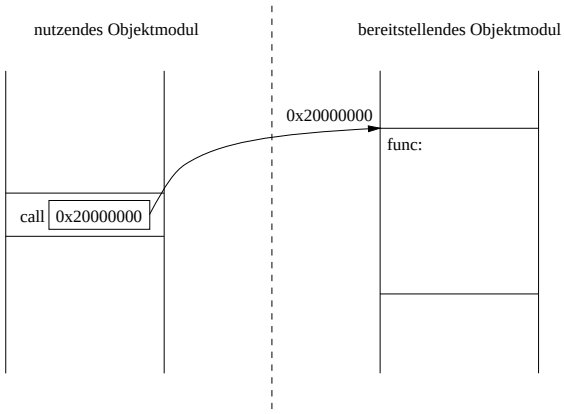


Abbildung 2.3: Situation nach Durchführung einer symbolischen Relokation

angegeben werden. Steht die tatsächliche Basisadresse des Objektmoduls fest, wird diese im Zug einer nichtsymbolischen Relokation zum Operanden addiert. Die tatsächliche Basisadresse ergibt sich, wenn Objektmodule vom Link-Editor zu einem statisch gelinkten Programm konkateniert wurden oder nachdem der Laufzeit-Linker ein Objektmodul in den Adreßraum eines Prozesses geladen hat.

## 2.6 Resolution

Treffen mehrere gleichnamige Symboldefinitionen aufeinander, so wird durch den Prozeß der *Resolution* mittels eines Satzes von Vorrangregeln entschieden, welche dieser Definitionen Verwendung findet. Die Resolution wird angewendet, wenn

- mehrere Objektmodule vom Link-Editor zu einer Ausgabedatei verbunden oder
- Objektmodule vom Laufzeit-Linker nach einer Symboldefinition durchsucht werden.

Auf Systemen, denen das Objektdateiformat ELF zugrunde liegt, basieren die Resolutionsregeln z. B. auf den folgenden Informationen:

1. **Art der Symbole**
   Ein definiertes Symbol hat Vorrang vor einem vorläufigen Symbol (d. h. eine initialisierte Variable wird statt einer gleichnamigen nichtinitialisierten Variablen verwendet). Ein vorläufiges Symbol hat Vorrang vor einem undefinierten Symbol.
2. **Bindungsart der Symbole**
   Ein lokal gebundenes Symbol hat Vorrang vor einem global gebundenen Symbol, das wiederum Vorrang vor einem schwach gebundenen Symbol hat.
3. **Herkunft der Symbole**
   Werden relozierbare und geteilt benutzbare Objektdateien durch den Link-Editor verarbeitet, so hat die Symboldefinition aus der relozierbaren Vorrang vor der Symboldefinition aus der geteilt benutzbaren Objektdatei.
4. **Reihenfolge der Symboldefinition**
   Eine zuerst aufgefundene Symboldefinition hat Vorrang vor allen anderen Symboldefinitionen. Diese Resolutionsregel wird auch als *Interposition* bezeichnet und speziell vom Laufzeit-Linker verwendet.

Vom Link-Editor werden die Regeln 1.–3. in obenstehender Reihenfolge verwendet. Kann keine der Regeln angewendet werden, so bricht der Link-Editor seine Arbeit mit einer Fehlermeldung („doppelt

definiertes Symbol“) ab. Der Laufzeit-Linker macht aus Performance-Gründen nur von den Regeln 2. und 4. Gebrauch.

Mit den in diesem Kapitel getroffenen Aussagen kann die Grundfunktion des Linkens als Vereinigung mehrerer Objektmodule und ihrer Symbolinformationen betrachtet werden. Hierbei sind die Resolutionsregeln zu berücksichtigen und abschließend Relokationen durchzuführen.

# Kapitel 3

# Konzepte des dynamischen Linkens

Grundprinzip des dynamischen Linkens ist es, symbolische Referenzen zwischen Objektmodulen erst zur Laufzeit aufzulösen. Damit wird der Erhalt symbolischer Nutzungsbeziehungen über die Phase der Programmentwicklung hinaus bis in die Phase der Programmausführung ermöglicht. Das Werkzeug, das symbolische Referenzen zur Laufzeit auflöst, wird als *Laufzeit-Linker* oder *dynamischer Linker* bezeichnet.

Durch das dynamische Linken werden Objektmodule nicht mehr wie beim statischen Linken in eine ausführbare Datei kopiert. Stattdessen referenziert eine dynamisch gelinkte Anwendung die von ihr benutzten Objektmodule. Auf diese Weise werden Objektmodule logischer, nicht aber physischer Bestandteil der Anwendung.

Der Begriff des *dynamischen Linkens* umfaßt damit

- die Erstellung dynamisch gelinkter Programme und geteilt benutzbarer Objektdateien durch den Link-Editor und
- die Auflösung symbolischer Referenzen zwischen diesen Modulen durch den Laufzeit-Linker (s. [Sun93a]).

Die folgenden Abschnitte beschreiben verschiedene Konzepte des dynamischen Linkens und erläutern, welche Anforderungen im Zusammenhang mit diesen Konzepten zu erfüllen sind.

## 3.1 Explizites dynamisches Linken

Als *explizites dynamisches Linken* soll verstanden werden, daß eine Anwendung zur Laufzeit die Dienste des dynamischen Linkers benutzt, um zusätzliche Objektmodule zu laden, zu relozieren und auf die darin enthaltenen Symbolwerte zuzugreifen. Zu diesem Zweck muß der dynamische Linker über eine Schnittstelle verfügen, die die folgenden, grundlegenden Operationen exportiert:

1. **Linken**
   Ein Objektmodul muß lokalisiert und in den Adreßraum eines Prozesses eingebracht werden. Im allgemeinen wird hierzu der Inhalt einer Objektdatei in den Hauptspeicher eingelesen. Sollen Objektmodule geteilt benutzt werden (s. Abschn. 3.4, S. 17), so muß in diesem Rahmen ein bereits in den Hauptspeicher eingelesenes Objektmodul verwendet werden können. In Abhängigkeit von der Art der geteilten Benutzung erfolgt anschließend die Relokation des Objektmoduls.
2. **Symbolzugriff**
   Ein Symbolwert (z. B. ein Datenobjekt oder eine Funktion) ist durch Angabe des Symbolnamens verfügbar zu machen. Im einfachsten Fall wird durch diese Operation die Hauptspeicheradresse des bezeichneten Objekts zurückgeliefert. Dem Nutzer obliegt es, diese Adresse in der richtigen Form zu interpretieren.

3. **Unlinken**
   Durch diese Operation wird ein nicht mehr benötigtes Objektmodul aus dem Adreßraum eines Prozesses entfernt. Hierbei gilt es zu beachten, daß ein Objektmodul nicht entfernt werden darf, solange andere geladene Objektmodule dessen Symbolwerte benutzen.

Die aufgezählten Operationen werden von den meisten dynamischen Linkern (z. B. [Ho91], [Engel95]) in Form einer Programmierschnittstelle bereitgestellt.

## 3.2 Implizites dynamisches Linken

Bei der Erzeugung dynamisch gelinkter Programme wird der Inhalt geteilt benutzbarer Objektdateien nicht in die zu erstellende Datei kopiert, sondern von dieser referenziert. Zu diesem Zweck werden vom Link-Editor die Namen der benötigten geteilt benutzbaren Objektdateien an gesonderter Stelle in der zu erzeugenden Datei vermerkt. Hierfür sind durch das Objektdateiformat geeignete Datenstrukturen zur Verfügung zu stellen (in Abschn. 5.4 auf S. 43 wird dies am Beispiel des ELF-Objektdateiformats ausführlich dargestellt). Beim Start eines dynamisch gelinkten Programms werden die betreffenden Objektmodule vom Laufzeit-Linker zu einer ausführbaren Einheit verbunden. Da die Aktivität hierbei nicht vom Programmcode, sondern vom Laufzeit-Linker ausgeht, soll dieser Mechanismus als *implizites dynamisches Linken* bezeichnet werden.

Zur Realisierung des impliziten dynamischen Linkens kann der Programmlader (z. B. `exec(2)`) um die Funktionalität des Laufzeit-Linkers erweitert werden. Eine weitere Möglichkeit besteht darin, den Programmlader so zu modifizieren, daß er ein dynamisch gelinktes Programm erkennt und in diesem Fall den Laufzeit-Linker als eigenständiges Modul lädt und vor dem Programmcode der Anwendung ausführt.

## 3.3 Abhängigkeiten von Objektmodulen

Durch das dynamische Linken ergeben sich zwei Arten von Abhängigkeiten zwischen Objektmodulen:

- **Benutzung von Symboldefinitionen**
  Verwendet der Laufzeit-Linker zur Auflösung symbolischer Referenzen eines Objektmoduls $A$ Symboldefinitionen eines Objektmoduls $B$, so ist $A$ von $B$ abhängig.

- **Referenzen zu Objektmodulen**
  Im Rahmen des impliziten dynamischen Linkens verweist ein Programm auf geteilt benutzbare Objektdateien, deren Symboldefinitionen es benötigt. Können geteilt benutzbare Objektdateien selbst wiederum dynamisch gelinkt sein, so verfügen auch sie u. U. über Referenzen zu weiteren geteilt benutzbaren Objektdateien.

Im allgemeinen wird die Benutzung von Symboldefinitionen eines Objektmoduls durch eine Referenz zu diesem Objektmodul ermöglicht. Aus diesem Grund wird im folgenden Text ein referenziertes Objektmodul als *Abhängigkeit* des referenzierenden Objektmoduls bezeichnet. Alle Abhängigkeiten eines Objektmoduls formen dessen *Abhängigkeitsliste*. Betrachtet man Objektmodule als Knoten und verbindet diese beginnend beim dynamisch gelinkten Programm durch gerichtete Kanten mit ihren Abhängigkeiten, so entsteht der sogenannte *Abhängigkeitsgraph*.

Zur Erklärung soll ein dynamisch gelinktes Programm $A$ dienen, daß von den Objektmodulen $B$ und $C$ abhängt, die wiederum beide von einem Objektmodul $D$ abhängig sind. Objektmodul $B$ hängt darüber hinaus von $A$ ab. Damit ergeben sich die Abhängigkeitslisten von $A$, $B$ und $C$ als $(B, C)$, $(A, D)$ und $(D)$. Die Abhängigkeitsliste von $D$ ist leer. Der sich aus dieser Nutzungsbeziehung ergebende Abhängigkeitsgraph ist in Abb. 3.1 dargestellt.

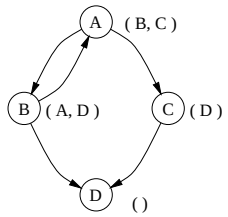


Abbildung 3.1: Beispiel eines Abhängigkeitsgraphen

Bei der Betrachtung von Abhängigkeitsgraphen verdienen mehrfach referenzierte Objektmodule und zyklische Abhängigkeiten besondere Beachtung:

- **mehrfach referenzierte Objektmodule**
  Mehrfach referenzierte Objektmodule sind in den Abhängigkeitslisten mehrerer Objektmodule eines Abhängigkeitsgraphen enthalten (z. B. Modul $D$ in Abb. 3.1).

- **zyklische Abhängigkeiten**
  Eine zyklische Abhängigkeit bezeichnet die wechselseitige Referenzierung zweier oder mehrerer Objektmodule (z. B. Module $A$ und $B$ in Abb. 3.1) und stellt somit einen Kreis im Abhängigkeitsgraphen dar. Zyklische Abhängigkeiten bedingen, daß eine Abhängigkeitsliste nicht rekursiv definiert sein darf, d. h. die Abhängigkeitsliste eines Objektmoduls darf nicht die Abhängigkeitslisten der von ihm referenzierten Module beinhalten.

Für mehrfach referenzierte Objektmodule und zyklische Abhängigkeiten ergibt sich das Problem, daß Objektmodule wiederholt geladen werden könnten. Im Fall zyklischer Abhängigkeiten besteht die Gefahr, daß der Laufzeit-Linker hierbei in eine Endlosschleife gerät. Zur Vermeidung dieser Probleme muß eine Überprüfung bereits geladener Objektmodule erfolgen. Zyklische Abhängigkeiten lassen sich vermeiden, wenn ein referenziertes Objektmodul die Symboldefinitionen des referenzierenden Objektmoduls verwenden kann. Sie sind in der Praxis von geringer Relevanz, da sie sich nicht (bzw. nur bewußt) erzeugen lassen. So kann z. B. Objektmodul $A$ aus Abb. 3.1 nicht mit einer Referenz zu Modul $B$ erzeugt werden, wenn $B$ nicht existiert und umgekehrt.

## 3.4 Geteilte Benutzung

Unter der *geteilten Benutzung* eines Objektmoduls ist zu verstehen, daß eine Hauptspeicherrepräsentation dieses Objektmoduls gleichzeitig von allen Anwendungen (Prozessen), die dieses Modul benötigen, verwendet werden kann. Die geteilte Benutzung von Objektmodulen trägt in entscheidendem Maß zur Verringerung des von mehreren Prozessen belegten Hauptspeichers bei.

Statisch gelinkte Programme benutzen mit Ausnahme von Betriebssystemdiensten nur Funktionen und Datenobjekte, die in ihnen selbst definiert sind. Sie verfügen über keine Nutzungsbezüge zu anderen Objektmodulen. Dadurch kann zwischen statisch gelinkten Programmen kein Objektmodul geteilt benutzt werden.

Durch das dynamische Linken wird die geteilte Benutzung von Objektmodulen ermöglicht, da diese als Bestandteil einer dynamisch gelinkten Anwendung identifizierbar und die Nutzungsbezüge zwischen ihnen nachvollziehbar sind.

Zur Auflösung symbolischer Referenzen zwischen geteilt benutzbaren Objektdateien und dynamisch gelinkten Programmen muß die Hauptspeicherrepräsentation beider Objektmodularten über Symbolinformationen verfügen[1].

Grundmechanismus für die geteilte Benutzung von Objektmodulen stellt gemeinsamer Speicher dar. Der Inhalt einer Objektdatei wird dabei physisch nur einmal in den Hauptspeicher eingelesen und

[1] Diese Informationen dürfen nicht wie im Fall statisch gelinkter Programme entfernt werden (z. B. mittels `strip(1)`).

dann in Form gemeinsamen Speichers in den Adreßraum der benutzenden Prozesse eingeblendet. Auf diese Weise wird ein Objektmodul logischer, nicht aber physischer Bestandteil eines Prozesses, wodurch die eingangs erwähnte Speichereinsparung ermöglicht wird. Es müssen Informationen verfügbar sein, die anzeigen, welche Objektmodule im Hauptspeicher zur geteilten Benutzung bereitstehen.

Zur Charakterisierung der geteilten Benutzung von Objektmodulen muß unterschieden werden, welche ihrer Komponenten geteilt benutzt werden können:

1. **Daten**
   Variable Datenobjekte gehören zum Status eines Prozesses und können daher nicht geteilt benutzt werden. Konstante Datenobjekte (z. B. Nur-Lese-Daten wie Texte von Status- oder Fehlermeldungen) können hingegen geteilt benutzt werden. Hierbei ist jedoch zu beachten, daß der Wert dieser Objekte Gegenstand einer Relokation sein kann, so z. B. der folgendermaßen definierte Wert von `s`:

   ```
   const char msg[] = "Hello World!";
   const char *s = msg;
   ```

2. **Anweisungen**
   Solange kein Prozeß den Anweisungsteil eines Objektmoduls modifiziert, kann er uneingeschränkt geteilt benutzt werden (s. [Kalfa88]). Meist wird der Anweisungsteil jedoch durch Relokationen an die Position des Objektmoduls im Adreßraum eines Prozesses angepaßt.

Die Modifikation konstanter Datenobjekte oder des Anweisungsteils eines Objektmoduls durch Relokationen schränkt die geteilte Benutzbarkeit (der entsprechenden Teile) des Objektmoduls ein. Sind diese Modifikationen für alle nutzenden Prozesse sichtbar, werden sie nur einmal innerhalb des ersten nutzenden Prozesses ausgeführt. Anhand der Adressierungsart der verwendeten Maschinensprache-Anweisungen können dabei die folgenden beiden Fälle unterschieden werden:

- **absolute Adressierung**
  Der Inhalt eines Objektmoduls $B$ werde von einem Objektmodul $A$ durch absolute Adressierung referenziert (s. Abb. 3.2). Durch den ersten Prozeß, der Objektmodul $A$ (und damit auch Objektmodul $B$) benutzt (Prozeß 1), wird die absolute Adresse auf einen Wert gesetzt, der von der Adresse von $B$ in Prozeß 1 abhängt ($Y$). Soll Objektmodul $A$ von einem weiteren Prozeß (Prozeß 2) benutzt werden, muß $B$ in Prozeß 2 an dieselbe absolute Adresse $Y$ wie in Prozeß 1 abgebildet werden.

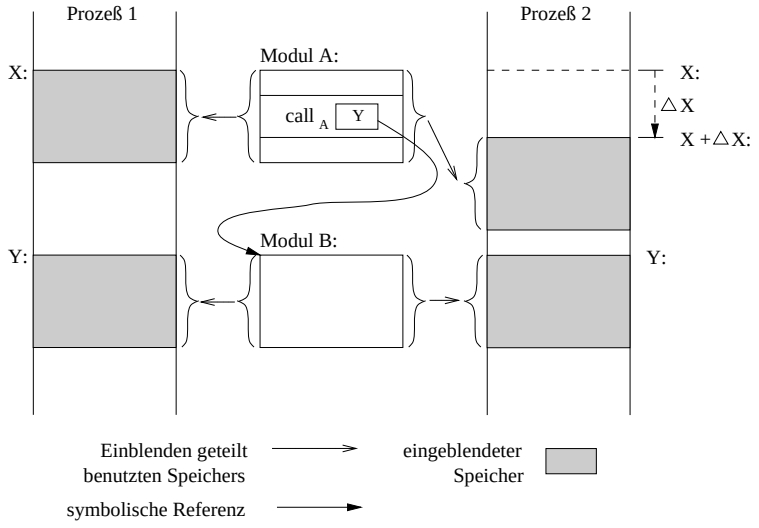


Abbildung 3.2: Geteilte Benutzung bei absoluter Adressierung

Da praktisch alle Objektmodule eigene Bestandteile referenzieren, hat diese Situation zur Folge, daß der Inhalt eines geteilt benutzten Objektmoduls in allen benutzenden Prozessen an derselben Adresse eingeblendet werden muß. Das erfordert, daß der Laufzeit-Linker über prozeßübergreifende Datenstrukturen verfügt, die ihm anzeigen, auf welche Adressen innerhalb eines Prozesses der Inhalt dieser Objektmodule einzublenden ist.

- **relative Adressierung**
  Der Inhalt eines Objektmoduls $B$ werde von einem Objektmodul $A$ durch relative Adressierung referenziert (s. Abb. 3.3). Durch den ersten Prozeß, der Objektmodul $A$ (und damit auch Objektmodul $B$) benutzt (Prozeß 1), wird die relative Adresse auf einen Wert gesetzt, der vom Abstand zwischen $A$ und $B$ in Prozeß 1 abhängt ($d$). Soll Objektmodul $A$ von einem weiteren Prozeß (Prozeß 2) benutzt werden, muß auch in Prozeß 2 derselbe Abstand $d$ zwischen den Modulen $A$ und $B$ eingehalten werden.

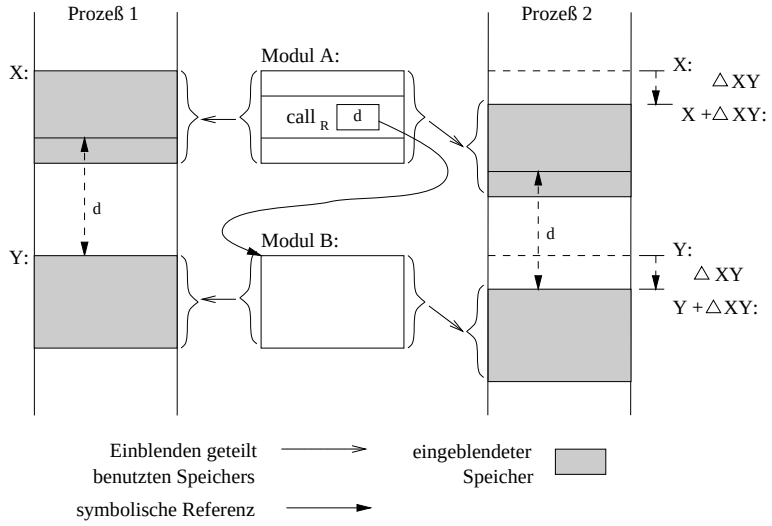


Abbildung 3.3: Geteilte Benutzung bei relativer Adressierung

Dem Laufzeit-Linker müssen somit prozeßübergreifende Datenstrukturen zur Verfügung stehen, die ihm anzeigen, welche Abstände im Adreßraum eines Prozesses zwischen geteilt benutzten Objektmodulen einzuhalten sind.

Sind die Modifikationen eines Objektmoduls durch Relokationen nicht für alle nutzenden Prozesse sichtbar, müssen die Relokationen für jeden nutzenden Prozeß wiederholt werden. Damit ergibt sich ein weiterer Fall zur Charakterisierung der geteilten Benutzung von Objektmodulen:

- **private Modifikation**
  Das Speichermanagement moderner Betriebssysteme erlaubt es, Modifikationen, die ein Prozeß an gemeinsamen Speicher ausführt, nur innerhalb dieses Prozesses sichtbar zu machen. Zu diesem Zweck erhält der Prozeß eine lokale Kopie der modifizierten Speichereinheit - z. B. der Speicherseite. Dieser Mechanismus wird auch als *copy-on-write policy* bezeichnet.

  Die geteilte Benutzung von Objektmodulen bei privater Modifikation gemeinsamen Speichers soll am Beispiel aus Abb. 3.4 dargestellt werden. Hierbei referenziert ein Objektmodul $A$ den Inhalt eines Objektmoduls $B$. Durch den ersten Prozeß, der Objektmodul $A$ (und damit auch Objektmodul $B$) benutzt (Prozeß 1), wird eine Relokation durchgeführt. Diese Relokation führt zu einer lokalen Kopie des Speicherabschnitts, der das Relokationsfeld enthält. Jeder weitere nutzende Prozeß (z. B. Prozeß 2) führt diese Relokation erneut aus und erhält daraufhin ebenso eine lokale Kopie des Speicherabschnitts.

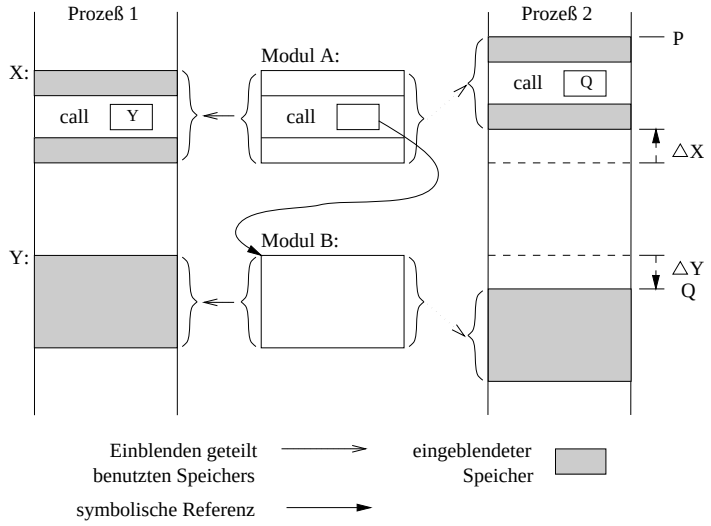


Abbildung 3.4: Geteilte Benutzung bei privater Modifikation

Für diesen Mechanismus muß die Hauptspeicherrepräsentation eines Objektmoduls neben Symbol- auch Relokationsinformationen beinhalten. Das Schema der privaten Modifikationen erlaubt die freie Wahl der Adresse, auf die ein geteilt benutzbares Objektmodul innerhalb eines Prozesses eingeblendet werden kann. Durch eine hohe Dichte von Relokationsfeldern werden jedoch viele lokale Kopien von Speicherbereichen erzeugt, was insgesamt eine Verschlechterung der geteilten Benutzbarkeit zur Folge hat. In Abschn. 5.8 (S. 45) wird ein Mechanismus vorgestellt, mit dem dieser Verschlechterung entgegengewirkt werden kann.

## 3.5 Typsicherheit

Durch das dynamische Linken werden symbolische Referenzen zwischen Objektmodulen erst zur Laufzeit aufgelöst. Da sich zwischen zwei Aufrufen einer dynamisch gelinkten Anwendung der Inhalt eines Objektmoduls ändern kann, ist es wichtig, daß zwischen Objektmodulen eine klare Übereinkunft über den Typ benötigter und bereitgestellter Symbole herrscht. Als *Typsicherheit* ist damit die Übereinstimmung des Typs einer Symbolreferenz und einer Symboldefinition zu verstehen. Typsicherheit wird durch den Vergleich von Typinformationen erreicht. Typinformationen beinhalten Angaben über

- den Typ eines Datenobjekts sowie
- den Typ des Rückgabewerts einer Funktion sowie Anzahl und Typen ihrer Parameter.

Auf der Seite eines bereitstellenden Objektmoduls gehen Typinformationen aus der Quelltext-Definition eines Datenobjekts oder einer Funktion hervor. Für ein nutzendes Objektmodul werden Typinformationen aus Prototypdeklarationen auf Quelltext-Ebene gewonnen. Besonders wichtig ist Typsicherheit für das Linken von Funktionen mit gleichem Namen und unterschiedlichen Parameterlisten (z. B. polymorphe Funktionen der Programmiersprache C++). Zur Auflösung symbolischer Referenzen reicht hierbei nicht mehr nur die Angabe des Quelltextnamens einer solchen Funktion aus, da diese erst durch Angabe ihrer Argumenttypen eindeutig identifiziert wird.

Eine Besonderheit bei der Gewinnung von Typinformationen von C/C++-Funktionen besteht in der Dualität ihrer Verwendung. Eine C/C++-Funktion kann im Sinne einer Programmiersprache wie Pascal sowohl in Form einer Funktion als auch einer Prozedur aufgerufen werden. Im Fall des Aufrufs in Prozedurform ist ohne vorherige Prototypvereinbarung[2] der Typ des Rückgabewerts nicht

[2] Dies führt bei den meisten Compilern lediglich zu einer Warnung, nicht jedoch zu einem Fehler.

ersichtlich. Aus diesem Grund muß für C/C++-Funktionen die Codierung des Typs des Rückgabewerts als Bestandteil der Typinformationen entfallen.

Zur Gewährleistung der Typsicherheit muß das Objektdateiformat ermöglichen, Symbolinformationen um Typinformationen zu ergänzen. Da dies jedoch nur bei wenigen Objektdateiformaten (z. B. HP-UX-a.out) der Fall ist, codieren C++-Compiler Typinformationen in den Symbolnamen und bilden somit die Übereinstimmung von Typinformationen auf die Übereinstimmung von Symbolnamen ab.

Wird der Laufzeit-Linker durch ein Kommando angewiesen, ein Datenobjekt oder eine Funktion verfügbar zu machen, so müssen die Regeln, die der C++-Compiler zur Namensbildung verwendet hat, bekannt sein.

## 3.6 Versionsverwaltung

Als *Versionsverwaltung* wird im Zusammenhang mit dem dynamischen Linken ein Bündel von Maßnahmen bezeichnet, durch das die folgenden Ziele in der angegebenen Reihenfolge verwirklicht werden sollen:

1. Kompatibilität von Objektmodulen
2. Verwendung der neuesten Version eines Objektmoduls

Die *Kompatibilität* kennzeichnet die Übereinstimmung von Definition und Verwendung von Elementen der Schnittstelle eines Objektmoduls. Als *Schnittstelle* soll hierbei die Menge der von einem Objektmodul bereitgestellten globalen Symboldefinitionen sowie deren Typinformationen und Wirkungsweise verstanden werden.

Grundlage einer Versionsverwaltung bildet die Kennzeichnung des Entwicklungsstands einer Schnittstelle (oder ihrer Elemente) mit einer *Versionsidentifikation* (z. B. Versionsnummer(n) o. Entwicklungsdatum). Die Versionsidentifikation muß zur Verwirklichung der beiden Ziele der Versionsverwaltung (s. o.) Informationen zur Kennzeichnung der Kompatibilitäts- und der Aktualitätsstufe eines Objektmoduls beinhalten. Die Wahrung der Kompatibilität wird durch die Übereinstimmung der Kompatibilitätsinformation von nutzendem und bereitstellendem Objektmodul erreicht. Zur Verwendung der neuesten Version eines Objektmoduls muß über der Aktualitätsinformation eine Ordnungsrelation definiert sein. Hierbei wird jeweils das Objektmodul verwendet, welches bezüglich dieser Relation die höchste Aktualitätsstufe besitzt.

## 3.7 Initialisierung und Terminierung globaler Datenobjekte

Die Initialisierung und Terminierung globaler Datenobjekte hat besonders mit der objektorientierten Programmierung an Bedeutung gewonnen. Durch die hierbei aufgerufenen Konstruktoren und Destruktoren können beliebig komplexe Aktionen ausgeführt werden.

Die Initialisierung eines globalen Objekts kann z. B. mit folgenden Operationen verbunden sein:

- Reservierung von Speicherplatz
- Öffnen einer Datei
- Aufbau einer Kommunikationsverbindung

Die Terminierung eines globalen Datenobjekts vollzieht meist die zur Initialisierung inversen Operationen, z. B.:

- Freigabe von Speicherplatz
- Schließen einer Datei

- Abbau einer Kommunikationsverbindung

Bei statisch gelinkten Programmen wird die Initialisierung und Terminierung globaler Datenobjekte durch die Laufzeitumgebung des Programms übernommen. Im Fall dynamisch gelinkter Anwendungen kann sich die Implementation benötigter geteilt benutzbarer Objektdateien ändern. Außerdem ist es möglich, durch dynamisches Linken auf Anforderung erst während des Programmlaufs Objektmodule in den Prozeß einzubinden. Dadurch ist es nicht mehr möglich, die Initialisierung und Terminierung globaler Datenobjekte von der Laufzeitumgebung des Programms durchführen zu lassen. Vielmehr muß jede geteilt benutzbare Objektdatei selbst die entsprechenden Anweisungen enthalten, die – gemäß ihrer Verwendung – im folgenden als *Initialisierungs-* und *Terminierungscode* bezeichnet werden sollen. Wenn ein Objektmodul dynamisch gelinkt wird, so ist der Initialisierungscode des Moduls durch den Laufzeit-Linker auszuführen. Durch den Laufzeit-Linker ist ebenfalls zu veranlassen, daß der Terminierungscode eines Objektmoduls ausgeführt wird, wenn der Inhalt des Objektmoduls aus dem Adreßraum des Prozesses entfernt oder der Prozeß beendet wird.

Der Initialisierungscode eines Objektmoduls $A$ muß auf die (initialisierten) globalen Datenobjekte der Objektmodule $B_1, B_2, \ldots$, von denen $A$ abhängt, zugreifen können. Hierfür ist es erforderlich, den Initialisierungscode der Module $B_1, B_2, \ldots$ vor dem des Moduls $A$ auszuführen. Daraus resultiert, daß die Abarbeitung des Initialisierungscodes bei den im Abhängigkeitsgraphen am weitesten unten stehenden Objektmodulen beginnt. Umgekehrt ist zu verhindern, daß der Terminierungscode des Objektmoduls $A$ auf die bereits terminierten globalen Datenobjekte der Module $B_1, B_2, \ldots$ zugreifen kann. Aus diesem Grund wird der Terminierungscode in umgekehrter Reihenfolge zum Initialisierungscode – also beginnend mit den im Abhängigkeitsgraphen am weitesten oben stehenden Objektmodulen – ausgeführt.

## 3.8 Verzögertes Binden

Durch das dynamische Linken bleiben symbolische Nutzungsbeziehungen über die Phase der Programmerzeugung hinaus bis zur Phase der Programmausführung erhalten. Das *verzögerte Binden* ist ein weiterführender Mechanismus, durch den symbolische Nutzungsbeziehungen erst zum Zeitpunkt ihrer tatsächlichen Benutzung aufgelöst werden. Anhand der Art des referenzierten Symbols werden zwei Fälle des verzögerten Bindens unterschieden:

1. **Verzögertes Binden von Funktionen**
   Ein Funktionsaufruf wird in einen Maschinenbefehl übersetzt (Kontrolltransferbefehl, z. B. `call`), dessen Operand ein Relokationsfeld darstellt. Zum Zeitpunkt des Starts eines dynamisch gelinkten Programms oder des Ladens einer geteilt benutzbaren Objektdatei wird dieses Relokationsfeld vom Laufzeit-Linker vorläufig so reloziert, daß bei Ausführung des Funktionsaufrufs die Steuerung an eine Routine des Laufzeit-Linkers übergeht. Diese übernimmt die endgültige Modifikation des Relokationsfelds mit der Adresse des Eintrittspunkts der Funktion und ruft anschließend die Funktion auf. Jede weitere Ausführung des Kontrolltransferbefehls überträgt von nun an die Steuerung direkt an die Funktion. Abschnitt 5.8.2 (S. 49) stellt eine konkrete Realisierung dieses Konzepts vor.

2. **Verzögertes Binden von Datenobjekten**
   Der Zugriff auf ein Datenobjekt wird in einen Maschinenbefehl übersetzt, von dem ein Operand mit der Adresse des Datenobjekts zu relozieren ist. Da es sich hierbei nicht um einen Kontrolltransferbefehl handelt, kann die Steuerung nicht wie im Fall des verzögerten Bindens von Funktionen an den Laufzeit-Linker übertragen werden. Auf den untersuchten Systemen wird das verzögerte Binden von Datenobjekten nicht realisiert.

Verzögertes Binden verhindert die Auflösung symbolischer Referenzen, die während des Programmlaufs nicht erreicht werden. Dadurch kann der Zugriff auf Teile von Objektmodulen (Symboltabelle u.a.), deren Symboldefinitionen unbenutzt bleiben, vermieden werden. In Abhängigkeit vom Speichermanagement und der Art des Zugriffs auf Dateien kann hierdurch das physische Lesen dieser Teile aus

den entsprechenden Objektdateien entfallen. Insgesamt kann durch das verzögerte Binden also der zeitliche Aufwand des dynamischen Linkens verringert werden. Das ist generell von Vorteil, hat aber auch zwei nachteilige Situationen zur Folge (s. [ATT91c]): Die erstmalige Ausführung eines verzögert gebundenen Funktionsaufrufs dauert länger als alle folgenden. Man kann sich Anwendungen vorstellen, die diese unterschiedlichen Ausführungszeiten nicht tolerieren können. Im Fall des dynamischen Linkens ohne verzögertes Binden wird das Fehlen einer Symboldefinition während des Programmstarts festgestellt. Durch das verzögerte Binden kann das zu einem späteren, unvorhersehbaren Zeitpunkt erfolgen.

## 3.9 Linkgranularität

Die *Linkgranularität* bezeichnet die kleinste Speichereinheit, die durch das dynamische Linken in den Adreßraum eines Prozesses eingefügt werden kann. In heutigen Systemen erfolgt das dynamische Linken meist auf der Ebene kompletter Objektmodule. Diese Vorgehensweise soll als *modulgranulares Linken* bezeichnet werden. Bei umfangreichen Objektmodulen erscheint diese Vorgehensweise auf den ersten Blick wenig effektiv, da auf diese Art eine Vielzahl von Symboldefinitionen zur Verfügung gestellt wird, von denen der betrachtete Prozeß u. U. nur einen geringen Teil benutzt[3]. Wird der Inhalt eines Objektmoduls jedoch von vielen Applikationen geteilt benutzt, ist insgesamt mit der Verwendung einer großen Anzahl von Symboldefinitionen zu rechnen. Die Effektivität des modulgranularen Linkens wird also durch geteilte Benutzung gesteigert. Es gilt dabei zu beachten, daß die Anzahl der Symboldefinitionen eines allgemeinen, geteilt benutzten Objektmoduls im Verhältnis zur Anzahl der nutzenden Prozesse stehen sollte.

Zur Adaption komplexer, objektorientierter Software-Systeme ist das Hinzufügen, Entfernen oder der Austausch von Datenobjekten und Funktionen erforderlich (s. [Schubert96]). Das Linken auf der Ebene dieser Elemente soll als *symbolgranulares Linken* bezeichnet werden. Hierfür ist es notwendig, einzelne Symbolwerte aus einem Objektmodul zu extrahieren und in den Adreßraum eines Prozesses einzufügen. Das setzt voraus, daß ein Symbolwert innerhalb des Objektmoduls durch seine Position und Größe beschrieben wird. Weiterhin ist zu beachten, daß ein Symbolwert möglicherweise weitere Symbolwerte referenziert. Als Beispiel soll eine Funktion `f()` dienen, durch deren Programmcode eine Funktion `g()` gerufen wird:

```
void f(){
  ...
  g();
  ...
}
```

Die Funktion `f()` ist damit von `g()` abhängig. Wird der Laufzeit-Linker angewiesen, `f()` zur Verfügung zu stellen, muß implizit auch `g()` als Abhängigkeit von `f()` geladen und mit `f()` verbunden werden. Ähnliches gilt für globale Datenobjekte. Hierfür sei folgendes Beispiel angegeben:

```
int a = 42;
int *b = &a;
```

Soll `b` verfügbar gemacht werden, so ist implizit auch der Wert von `a` als Abhängigkeit von `b` zu laden. Anschließend muß die Initialisierung von `b` mit der Adresse von `a` erfolgen.

Zur Feststellung der Abhängigkeiten von Symbolwerten ist es vorstellbar, Relokationsinformationen zu benutzen. Soll ein Symbolwert zur Verfügung gestellt werden, wird hierzu untersucht, welche Relokationsfelder im entsprechenden Speicherbereich liegen. Anhand der dazugehörigen Relokationsinformationen kann auf die benötigten Symbolwerte geschlossen werden. Abbildung 3.5 stellt dies am Beispiel der oben erwähnten Funktionen `f()` und `g()` dar. Sind Relokationsfelder jedoch nicht Be-

[3] Als Beispiel sei hier auf `libc.so.1` verwiesen, die unter Solaris 2.x mehr als 1000 globale Funktionen und Datenobjekte definiert.

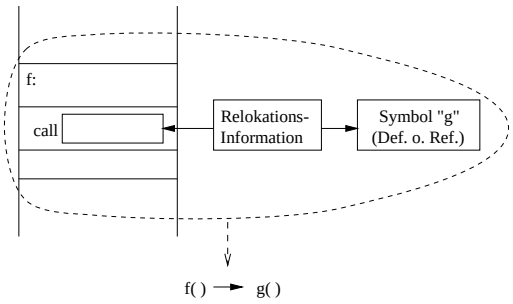


Abbildung 3.5: Abhängigkeiten von Symbolwerten

standteil des Speicherbereichs eines Symbolwerts (s. Abschn. 5.8, S. 45), wird diese Vorgehensweise hinfällig.

Für eine Verwendung globaler Datenobjekte ist ggf. Initialisierungs- und Terminierungscode notwendig. Symbolgranulares Linken erfordert also Informationen über Abhängigkeiten sowie Initialisierungs- und Terminierungscode von Symbolwerten. Durch heutige Übersetzungssysteme werden diese Informationen nicht in Objektmodule integriert (bzw. können vom zugrundeliegenden Objektdateiformat nicht gespeichert werden). Aus diesem Grund kann symbolgranulares dynamisches Linken nicht realisiert werden.

# Kapitel 4

# Das ELF-Objektdateiformat

Mit dem Wechsel von UNIX System V Release 3 zu Release 4 wurde das bis dahin verwendete COFF-Objektdateiformat (Common Object File Format) durch das ELF-Objektdateiformat (Executable and Linking Format) ersetzt (s. [ATT91b]). ELF wurde ursprünglich als Bestandteil des System V Application Binary Interface (ABI) der UNIX System Laboratories definiert, stellt heute jedoch ein systemübergreifendes Format für Objektdateien dar (s. [TIS94]).

Mit ELF ist ein modernes, flexibles Objektdateiformat geschaffen worden, das in seiner Struktur weitestgehend unabhängig von der zugrundeliegenden Architektur ist. Durch ELF werden Konzepte des dynamischen Linkens, so z. B. geteilt benutzbare Objektmodule oder Initialisierungs- und Terminierungscode, unterstützt. Mit der zu erwartenden Migration vieler Systeme hin zur Konformität mit UNIX System V Release 4 ist mit einer großen Verbreitung von ELF zu rechnen.

## 4.1 Aufbau einer ELF-Datei

Objektdateien werden vom Link-Editor zur Programmerzeugung und vom Laufzeit-Linker sowie dem Programmlader zur Programmausführung verwendet. Entsprechend dieser beiden Aufgaben wird der Inhalt einer ELF-Datei in *Sektionen* und *Segmente* strukturiert (s. Abb. 4.1).

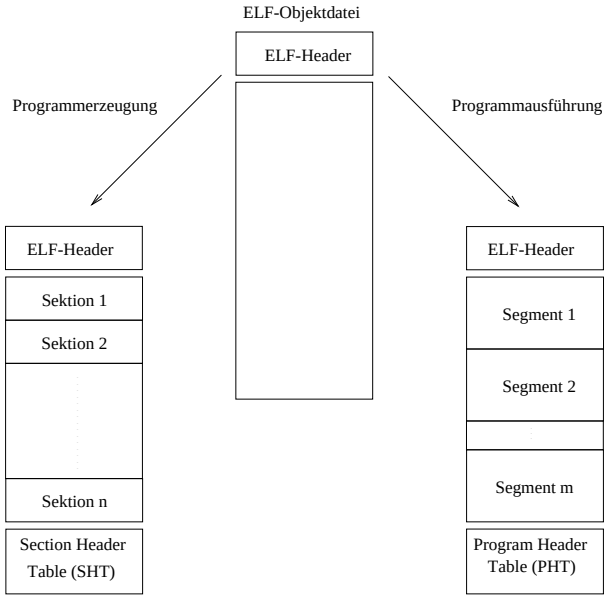


Abbildung 4.1: Aufbau einer ELF-Objektdatei

Beide Strukturen ermöglichen für jede der Aufgaben den möglichst effektiven Zugang zu den jeweils benötigten Informationen der ELF-Datei.

Der Inhalt von ELF-Dateien, die vom Link-Editor als Eingabe verarbeitet werden können (relozierbare bzw. geteilt benutzbare Objektdateien), ist in Sektionen gegliedert. Jede Sektion wird durch einen Eintrag in der *Section Header Table* (SHT) beschrieben. Der Inhalt ladbarer ELF-Dateien (statisch bzw. dynamisch gelinkte Programme und geteilt benutzbare Objektdateien) ist in *Segmente* gegliedert. Segmente werden durch Einträge in der *Program Header Table* (PHT) beschrieben. Je nach Verwendungszweck kann eine ELF-Datei in Sektionen und Segmente oder nur in Sektionen oder Segmente unterteilt werden.

Die Positionen von Komponenten ladbarer ELF-Dateien werden durch relative Adressen bezeichnet. Zu diesen Komponenten zählen z. B.

- Symbolwerte,
- Relokationsfelder,
- Hash-, Relokations-, String- und Symboltabellen (s. u.).

Relative Adressen beziehen sich auf die *Basisadresse* der ELF-Datei. Während der Erzeugung einer ladbaren ELF-Datei wird von einer Basisadresse 0 ausgegangen, zur Laufzeit ergibt sich die Basisadresse aus der Position im Adreßraum eines Prozesses, an die der Inhalt der Datei geladen wurde. Komponenten relozierbarer Objektdateien werden hingegen durch ihre Position innerhalb der Datei (*File Offset*) oder innerhalb einer Sektion (*Section Offset*) beschrieben.

Jede ELF-Datei enthält einen *Header*. Der Header ist eine Struktur mit globalen Informationen über die ELF-Datei. Er ist die einzige Komponente einer ELF-Datei mit fixer Position. Sektionen, Segmente, die Section Header Table und die Program Header Table können innerhalb einer ELF-Objektdatei frei angeordnet werden.

## 4.2 Header

Der Header befindet sich stets am Anfang einer ELF-Datei und enthält globale Informationen, die den Zugang zu ihrem Inhalt ermöglichen und ihre Verwendbarkeit anzeigen. Der ELF-Header beginnt mit einer Identifikations-Zeichenkette, die folgende Informationen enthält:

- **Magic-Number**
  Die Magic-Number ist eine charakteristische Zeichenfolge, die dazu dient, eine Datei als ELF-Datei zu identifizieren und damit von Dateien anderer Formate zu unterscheiden (s. `/etc/magic`).
- **Wortbreite**
  Damit wird die der Datei zugrundeliegende Wortbreite verschlüsselt (Gegenwärtig ist ELF für eine Wortbreite von 32 Bits spezifiziert, eine Umstellung auf eine andere Wortbreite ist jedoch ohne größere Probleme möglich.).
- **Art der Datenkodierung**
  ELF unterscheidet zwischen LSB- (least significant byte first) und MSB- (most significant byte first) Kodierung.
- **ELF-Version**
  Mit diesem Feld wird die der Datei zugrundeliegende ELF-Version kodiert. Mit Hilfe der Version kann über die Kompatibilität der ELF-Datei und der Routinen, die auf diese Datei zugreifen, entschieden werden.

Die Identifikations-Zeichenkette ermöglicht es, unabhängig von Wortbreite und Datenkodierung Strukturinformationen über die ELF-Datei zu erlangen. Mit diesen Informationen ist eine Interpretation der folgenden, ebenfalls im Header enthaltenen Informationen möglich:

- **Dateityp**
  ELF unterscheidet relozierbare, geteilt benutzbare und ausführbare Objektdateien. Darüber hinaus kann eine Objektdatei auch als core dump (`core`) ausgewiesen werden.

- **Architektur**
  Durch dieses Attribut wird die dem Inhalt der Datei zugrundeliegende Architektur (z. B. SPARC, Intel 80386 oder Motorola 68000) angegeben.

- **Eintrittspunkt**
  Handelt es sich um eine ausführbare Datei (statisch oder dynamisch gelinktes Programm), so wird mit dem Eintrittspunkt die Startadresse des Programms angegeben. Dieses Feld enthält den Wert 0, wenn die Datei nicht ausführbar ist (relozierbare oder geteilt benutzbare Objektdatei).

Da der Header der einzige Teil einer ELF-Datei mit fixer Position ist, muß er den Zugang zu allen anderen Komponenten ermöglichen. Zu diesem Zweck beschreibt er sowohl die SHT als auch die PHT durch Anfangsposition sowie Größe und Anzahl ihrer Einträge.

## 4.3 Sektionen

Sektionen stellen einen zusammenhängenden Bereich einer ELF-Datei dar. Jede Sektion wird durch genau einen SHT-Eintrag – den *Section Header* – beschrieben. Es gibt Section Header, denen keine Sektion zugeordnet ist. Sektionen dürfen sich nicht überlappen. Nicht der gesamte Bereich einer ELF-Datei muß durch Sektionen abgedeckt werden, d. h. zwischen zwei Sektionen können Zwischenräume existieren, deren Inhalt unspezifiziert ist. Durch Zwischenräume können Ausrichtungsanforderungen einzelner Sektionen erfüllt werden.

Jeder Section Header (und damit jede Sektion) wird durch seinen SHT-Index eindeutig identifiziert. Eine besondere Rolle spielt dabei der SHT-Eintrag 0. Bezüge auf diesen SHT-Eintrag repräsentieren undefinierte oder fehlende Informationen. So werden z. B. undefinierte Symbole durch einen Bezug auf diesen Eintrag notiert. Darüber hinaus existiert eine Reihe weiterer reservierter Indizes, für die es weder einen SHT-Eintrag noch eine Sektion gibt. So werden z. B. vorläufige und absolute Symbole (s. Abschn. 2.4.2, S. 9) durch einen Bezug auf einen reservierten SHT-Index gekennzeichnet.

Der Section Header beschreibt eine Sektion durch folgende Angaben:

- **Name**
  Der Name wird durch einen Index in eine spezielle Sektion, die *Section Header String Table*, gekennzeichnet. Da diese spezielle Sektion nicht durch ihren Namen (`.shstrtab`) identifiziert werden kann (dazu wäre sie selbst nötig), enthält der ELF-Header ihren SHT-Index. Namen von Sektionen mit vordefinierter Bedeutung beginnen mit einem Punkt (z. B. `.text`, `.data`, `.symtab`).

- **Typ**
  Durch den Typ werden Inhalt und Verwendungszweck einer Sektion näher beschrieben. Er kennzeichnet eine Sektion z. B. als Symboltabelle, Relokationstabelle, String- oder Hash-Tabelle.

- **Flags**
  Durch entsprechende Flags wird angezeigt, ob der Inhalt der Sektion in den Hauptspeicher geladen werden soll, beschreibbar und/oder ausführbar ist.

- **Position**
  Die Position zeigt den Beginn einer Sektion in der Datei an. Soll eine Sektion als Bestandteil eines Objektmoduls in den Hauptspeicher geladen werden, so gibt der Section Header zusätzlich deren relative Adresse an.

- **Größe**
  Mit diesem Feld wird die Größe der Sektion angegeben. Es gibt Sektionen, die keinen Platz in der Datei, wohl aber im Hauptspeicher einnehmen und daher in der Datei nur durch den Section Header vertreten sind. Die Größe gibt in diesem Fall an, wieviel Platz für diese Sektion zur Laufzeit im Hauptspeicher zu reservieren ist. Ein Beispiel dafür ist die Sektion für nichtinitialisierte Daten (`.bss`-Sektion, s. Tabelle 4.1).

- **Ausrichtungsanforderung**
  Durch dieses Attribut wird angegeben, auf welche Speichergrenze eine Sektion auszurichten ist.

- **Größe eines Eintrags**
  Ist der Inhalt einer Sektion strukturiert (z. B. die Symboltabelle), so wird die Größe eines Eintrags angegeben.

Darüber hinaus enthält ein Section Header Informationen, die zur Verknüpfung von Sektionen verwendet werden. So verweist der Section Header einer Relokationstabelle z. B. auf zwei weitere Sektionen: die Sektion, die durch die Relokationen verändert wird, und die Symboltabelle, die die von den symbolischen Relokationen benötigten Symbole enthält.

Tabelle 4.1 enthält eine Zusammenstellung der Sektionen, die speziell für das Linken von Interesse sind.

## 4.4 Segmente

Beim Laden einer ELF-Datei wird nicht die Sektions-, sondern die Segmentstruktur benutzt. Segmente sind zusammenhängende Bereiche der Objektdatei. Anders als Sektionen können Segmente gleiche Bereiche innerhalb der Datei bezeichnen. So ist es z. B. möglich, Segmente ineinander zu schachteln. Segmente müssen nicht den gesamten Bereich einer Objektdatei abdecken. Zwischen Segmenten können Zwischenräume existieren (z. B. um Ausrichtungsanforderungen gerecht zu werden). Ein Segment vereint meist mehrere Sektionen mit gleichen Hauptspeicher-Zugriffsrechten. Jedes Segment wird durch genau einen PHT-Eintrag – den *Program Header* – beschrieben.

Der Program Header beschreibt ein Segment durch folgende Angaben:

- **Typ**
  Der Typ kennzeichnet Art und Verwendung des Segments (s. folgende Abschn.).

- **Position**
  Es wird der Beginn eines Segments in der Datei und ihre relative Adresse im Hauptspeicher angegeben. Program Header sind in der PHT nach aufsteigender relativer Adresse des Segments geordnet.

- **Größe**
  Es wird die tatsächliche Größe des Segments innerhalb der Datei ($F$) und die gewünschte Hauptspeichergröße ($M$) angegeben. Beim Laden des Segments werden stets $M$ Bytes im Hauptspeicher reserviert. Gilt $M > F$, so werden die zusätzlichen $M - F$ Bytes mit dem Wert 0 initialisiert. Der Fall $F > M$ darf hingegen nicht auftreten.

- **Flags**
  Mittels dieses Attributs werden die Zugriffsrechte auf den Inhalt eines Segments festgelegt. Der Inhalt eines Segments kann als ausführbar, lesbar und/oder beschreibbar gekennzeichnet werden. Die tatsächlichen Zugriffsrechte sind vom Speichermanagement des Betriebssystems abhängig. Alle Kombinationen der aufgeführten Flags sind erlaubt. Das System darf u. U. mehr Zugriffsrechte vergeben, als durch die Flags festgelegt sind (so kann z. B. das Lese- im Schreibrecht enthalten sein), lediglich eine fehlende Schreibberechtigung darf nicht ignoriert werden.

| Name der Sektion | Inhalt der Sektion |
|---|---|
| `.bss` | nichtinitialisierte Daten<br>(Diese Sektion belegt keinen Platz in der ELF-Datei, d. h. es existiert nur der Section Header, der die Größe des Speicherbereichs angibt, der zur Laufzeit für nichtinitialisierte Daten reserviert werden muß.) |
| `.data` | initialisierte variable Daten |
| `.dynamic` | Informationen für das dynamische Linken |
| `.dynstr` | Stringtabelle für das dynamische Linken |
| `.dynsym` | Symboltabelle für das dynamische Linken |
| `.fini` | Terminierungscode einer Objektdatei<br>(s. Abschn. 5.5, S. 43) |
| `.got` | Global Offset Table<br>(s. Abschn. 5.8.1, S. 46) |
| `.hash` | Hashtabelle<br>(s. Abschn. 4.7, S. 32) |
| `.init` | Initialisierungscode einer Objektdatei<br>(s. Abschn. 5.5, S. 43) |
| `.interp` | Name des Programminterpreters<br>(Ein Beispiel für einen Programminterpreter ist in diesem Zusammenhang der Laufzeit-Linker.) |
| `.plt` | Procedure Linkage Table (PLT)<br>(s. Abschn. 5.8.2, S. 49) |
| `.rel`*name*<br>`.rela`*name* | Relokationstabelle<br>(*name* ist durch den Namen der Sektion zu ersetzen, auf die sich die Relokationen beziehen, für die `.text`-Sektion etwa heißt die Relokationstabelle `.rel.text` bzw. `.rela.text`) |
| `.rodata` | initialisierte konstante Daten (Nur-Lese-Daten) |
| `.shstrtab` | Stringtabelle mit den Sektionsnamen |
| `.strtab` | Stringtabelle mit Symbolnamen |
| `.symtab` | Symboltabelle |
| `.text` | Programmcode |

Tabelle 4.1: Sektionen einer ELF-Datei

- **Ausrichtungsanforderung**
  Durch dieses Attribut kann ein Segment auf eine bestimmte Adresse (z. B. auf eine Seitengrenze) ausgerichtet werden.

Die Gliederung einer Objektdatei in Segmente soll einen möglichst effizienten Ladeprozeß ermöglichen. Die Segmente einer ELF-Datei werden in Hauptspeichersegmente des Prozesses abgebildet. Da der Laufzeit-Linker den Inhalt geteilt benutzbarer Objektdateien in den Adreßraum eines Prozesses laden muß, ist die Segmentstruktur einer ELF-Objektdatei von entscheidender Bedeutung für das dynamische Linken.

### 4.4.1 Textsegment

Ein Segment, dessen Inhalt gelesen und ausgeführt werden kann (s. Attribute des Program Header), wird typischerweise als *Textsegment* bezeichnet. Das Textsegment umfaßt meist die in Abb. 4.2 dargestellten Sektionen.

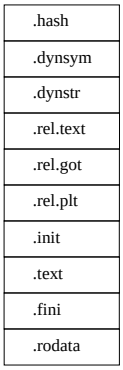


Abbildung 4.2: Textsegment

### 4.4.2 Datensegment

Ein Segment, dessen Inhalt gelesen und beschrieben werden kann, wird typischerweise als *Datensegment* bezeichnet. Das Datensegment besteht meist aus den in Abb. 4.3 dargestellten Sektionen.

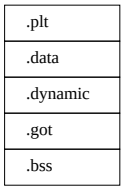


Abbildung 4.3: Datensegment

Die in Abb. 4.3 gezeigte `.bss`-Sektion stellt zur Laufzeit den Hauptspeicherplatz für nichtinitialisierte Variablen zur Verfügung. Diese Sektion soll keinen Platz in der Datei belegen. Durch ihre Anordnung am Ende des Datensegments und durch unterschiedliche Angaben für Datei- und Hauptspeichergröße des Segments (Dateigröße $<$ Hauptspeichergröße) wird erreicht, daß erst zur Laufzeit Platz im Hauptspeicher für die `.bss`-Sektion reserviert wird.

### 4.4.3 Dynamic-Segment

Jedes dynamisch gelinkte Programm und jede geteilt benutzbare Objektdatei enthält ein spezielles Segment mit Informationen für das dynamische Linken. Dieses Segment beinhaltet die `.dynamic`-Sektion (s. Tabelle 4.1) und soll im weiteren Verlauf als *Dynamic-Segment* bezeichnet werden.

Das Dynamic-Segment ist ein Feld, durch dessen Einträge die folgenden Elemente eines dynamisch gelinkten Programms bzw. einer geteilt benutzbaren Objektdatei beschrieben werden:

- **Benötigte geteilt benutzbare Objektdateien**
  Die Namen benötigter geteilt benutzbarer Objektdateien werden durch einen Index in die Stringtabelle für das dynamische Linken angegeben. Die Ordnung dieser Angaben bestimmt die Ladereihenfolge der entsprechenden geteilt benutzbaren Objektdateien.

- **Suchpfad für benötigte geteilt benutzbare Objektdateien**
  In der Stringtabelle für das dynamische Linken kann eine Liste von Pfadangaben enthalten sein, die vom Laufzeit-Linker zur Lokalisierung der benötigten Objektdateien benutzt wird. Dieser sogenannte *Runpath* wird durch einen Index in die Stringtabelle spezifiziert.

- **Relokationsinformationen**

Relokationstabellen werden durch ihre (relative) Adresse, ihre Größe und die Größe ihrer Einträge beschrieben. Durch das Dynamic-Segment können drei Relokationtabellen bezeichnet werden: für Relokationen mit und ohne explizitem Zusatz (s. Abschn. 4.8, S. 33) sowie für PLT-Relokationen (s. Abschn. 5.8.2, S. 49).

- **Hashing-Informationen für die Symbolsuche**
  Die Hashtabelle (s. Abschn. 4.7, S. 32) wird durch ihre relative Adresse beschrieben.
- **Stringtabelle für das dynamische Linken**
  Die relative Adresse und die Größe der Stringtabelle für das dynamische Linken (Sektion `.dynstr`, s. Abschn. 4.5, S. 31) werden angegeben.
- **Symboltabelle für das dynamische Linken**
  Die Symboltabelle für das dynamische Linken (Sektion `.dynsym`, s. Abschn. 4.6, S. 32) wird durch ihre relative Adresse und die Größe ihrer Einträge beschrieben.
- **Initialisierungs- und Terminierungscode**
  Initialisierungs- und Terminierungscode wird durch die relative Adresse der entsprechenden Funktionen bezeichnet (s. Abschn. 5.5, S. 43).
- **Versionsbehafteter Dateiname**
  Durch einen Index in die Stringtabelle für das dynamische Linken kann der versionsbehaftete Name der betrachteten geteilt benutzbaren Objektdatei angegeben werden (s. Abschn. 5.9, S. 51).

Darüber hinaus können im Dynamic-Segment weitere Informationen enthalten sein, die das Verhalten des Laufzeit-Linkers beeinflussen. So kann z. B. das Vorhandensein von Textrelokationen angezeigt oder die Reihenfolge der Symbolsuche innerhalb dynamisch gelinkter Anwendungen verändert werden.

## 4.5 Stringtabellen

Folgende Elemente einer ELF-Objektdatei werden durch Zeichenketten repräsentiert:

- Symbolnamen
- Sektionsnamen
- der Suchpfad für benötigte geteilt benutzbare Objektdateien
- Namen benötigter geteilt benutzbarer Objektdateien
- der versionsbehaftete Name einer geteilt benutzbaren Objektdatei

Diese Zeichenketten werden innerhalb einer ELF-Datei in Stringtabellen gespeichert und durch einen Index identifiziert. Eine Stringtabelle ist meist in einer eigenen Sektion der Datei untergebracht. Zeichenketten werden innerhalb einer Stringtabelle durch Null-Zeichen (`'\0'`) voneinander getrennt. Eine Stringtabelle beginnt stets mit einem Null-Zeichen. Damit ist ein Stringtabellen-Index 0 gleichbedeutend mit keinem oder einem undefinierten Namen.

Durch die Verwendung von Stringtabellen kann der Name einer Datenstruktur (Symbol, Sektion) als Index dargestellt und in der Struktur abgespeichert werden. Damit können Strukturen fester Größe über (nahezu) beliebig lange Namen verfügen. Dies ist besonders für die Codierung von Typinformationen in Symbolnamen von Bedeutung.

Die Sektion `.shstrtab` beinhaltet die Stringtabelle mit den Sektionsnamen, die Sektion `.strtab` enthält die Stringtabelle mit den Namen der Symbole, die in der Symboltabelle für das statische Linken (`.symtab`) enthalten sind. Von besonderer Bedeutung für das dynamische Linken ist die Sektion `.dynstr`. Dynamisch gelinkte Programme und geteilt benutzbare Objektdateien verfügen über eine

solche Sektion. Sie enthält die Namen aller Symbole, die für das dynamische Linken benutzt werden (Symboltabelle `.dynsym`), die Namen benötigter geteilt benutzbarer Objektdateien sowie den Suchpfad für deren Lokalisierung. Im Fall einer geteilt benutzbaren Objektdatei kann die `.dynstr`-Sektion außerdem den versionsbehafteten Namen dieser Datei enthalten.

## 4.6 Symboltabellen

Die Symboltabelle einer ELF-Datei beschreibt Symboldefinitionen und -deklarationen (undefinierte Symbole) durch folgende Attribute:

- **Name**
  Dieses Feld identifiziert den Symbolnamen durch einen Index in die entsprechende Stringtabelle (`.strtab` oder `.dynstr`). Ist der Index 0, so hat das Symbol keinen Namen.
- **Wert**
  Handelt es sich um ein vorläufiges Symbol, so verkörpert der Wert eine Ausrichtungsanforderung. Für definierte Symbole stellt der Wert im Fall einer relozierbaren Objektdatei die Position innerhalb der Sektion dar, die das repräsentierte Datenobjekt oder die repräsentierte Funktion beinhaltet. Im Fall von dynamisch zu linkenden Objektdateien wird damit die relative Adresse des repräsentierten Elements angegeben.
- **Größe**
  Dieses Attribut kennzeichnet die Größe des durch den Symbolwert belegten (oder benötigten) Speicherbereichs.
- **Bindungsart**
  Dieses Attribut zeigt an, ob das Symbol global, lokal oder schwach gebunden ist (s. Abschn. 2.4.4, S. 10).
- **Typ**
  ELF unterscheidet zwischen Symboltypen zur Repräsentation von Datenobjekten, Funktionen, Sektionen und Dateinamen.

  Symbole zur Repräsentation von Sektionen bezeichnen die relative Adresse einer Sektion, die in den Speicher geladen wird (= Element eines Segments ist).

  Symbole zur Repräsentation von Dateinamen geben meist Namen von Quelltextdateien an, die zur Erstellung der Objektdatei verwendet wurden.
- **SHT-Index**
  Diesem Index kommt eine besondere Bedeutung bei: Im Fall einer Symboldefinition wird damit die Sektion bezeichnet, die das repräsentierte Element enthält. Reservierte Indizes (s. Abschn. 4.3, S. 27) hingegen markieren undefinierte, vorläufige oder absolute Symbole.

Die Sektion `.symtab` enthält die Tabelle aller Symbole einer ELF-Datei. Davon werden jedoch nicht alle für das dynamische Linken benötigt (z. B. Symbole zur Repräsentation von Dateinamen). Aus diesem Grund existiert speziell für das dynamische Linken eine in der Sektion `.dynsym` enthaltene Symboltabelle. Durch Informationen im SHT-Eintrag einer Symboltabelle wird auf die Stringtabelle verwiesen, die die Namen der Symbole enthält (für `.symtab` ist das `.strtab`, für `.dynsym` wird `.dynstr` verwendet).

## 4.7 Hashtabelle

Dynamisches Linken beeinflußt die Zeit, die für Programmstart und -ausführung benötigt wird. Aus diesem Grund muß die Zeit, die zur Suche einer Symboldefinition notwendig ist, möglichst kurz gehalten werden. Zu diesem Zweck enthält jede dynamisch gelinkte Objektdatei eine Hashtabelle, durch

die der Zugriff auf die für das dynamische Linken verwendete Symboltabelle (Sektion `.dynsym`) beschleunigt wird.

Die Hashtabelle ist im Fall geteilt benutzbarer Objektdateien oder dynamisch gelinkter Programme in der Sektion `.hash` enthalten. Die Menge der Symbole aus `.dynsym` wird gemäß des Resultats der Anwendung einer Hashing-Funktion $f()$ auf den Symbolnamen *symname* in Untermengen (*buckets*) zerlegt. Bei der Suche nach einem Symbol braucht somit nur die durch $f(symname)$ bezeichnete Untermenge anstatt der gesamten Symbolmenge durchsucht werden.

## 4.8 Relokationstabellen

Im Abschn. 2.5 (S. 11) wurde die Relokation als Prozeß vorgestellt, der Relokationsfelder mit absoluten Adressen aktualisiert. Relokationen werden durch Relokationsinformationen gesteuert. Relokationsinformationen unter ELF sind ein Quadrupel aus der relativen Adresse des Relokationsfelds, einem Symboltabellenindex (optional), dem Relokationstyp und einem expliziten Zusatz (optional).

Unter einem *expliziten Zusatz* (*addend*) ist ein Wert zu verstehen, der zur absoluten Adresse addiert wird. Werden Relokationsinformationen ohne expliziten Zusatz verwendet, bildet der Inhalt des Relokationsfelds selbst einen *impliziten Zusatz*. Durch Verwaltung expliziter oder impliziter Zusätze wird ELF architekturspezifischen Anforderungen an Relokationen gerecht[1].

Der Relokationstyp legt fest, in welcher Weise die zur Berechnung der absoluten Adresse verwendeten Werte miteinander verknüpft werden und gibt außerdem an, welche Teile (Bits) des Relokationsfelds zu modifizieren sind. Auf diese Art und Weise kann man den verschiedenen Befehlsformaten und Adressierungsmodi der einzelnen Prozessoren gerecht werden. Architekturabhängig existiert meist eine Vielzahl von Relokationstypen[2], hierzu sei auf spezielle Literatur wie [Sun93a], [TIS94] oder [ATT91a] verwiesen.

Für symbolische Relokationen muß die Relokationsinformation den Index des Symboltabelleneintrags beinhalten, der den Namen des benötigten Symbols enthält.

Die Durchführung von Relokationen soll am Beispiel der in Tabelle 4.2 dargestellten Werte erläutert werden.

| Kürzel | Wert |
|---|---|
| `a` | impliziter oder expliziter Zusatz |
| `b` | Basisadresse eines Objektmoduls zur Laufzeit |
| `f` | absolute Adresse eines Relokationsfelds zur Laufzeit |
| `r` | Adresse des Relokationsfelds bezüglich der Basisadresse des Objektmoduls (relative Adresse), ergibt sich bei der Erzeugung eines Objektmoduls |
| `s` | absolute Adresse eines Symbolwerts zur Laufzeit, Ergebnis der Symbolsuche |

Tabelle 4.2: Größen zur Berechnung des Relokationswerts

Die absolute Adresse eines Relokationsfelds zur Laufzeit ergibt sich aus

```
f = b + r;
```

Eine symbolische Relokation wird anschließend mittels

```
*f = s;
```

[1] Auf SPARC und Intel i860 werden z. B. ausschließlich Relokationsinformationen mit, auf Intel 80386 hingegen ausschließlich Relokationsinformationen ohne expliziten Zusatz verwendet.

[2] Auf SPARC werden z. B. mehr als 20 Relokationstypen unterschieden.

durchgeführt. Die absolute Adresse (Laufzeit-Adresse) eines im selben Objektmodul befindlichen Elements kann durch eine nichtsymbolische Relokation bestimmt werden. Hierzu wird während der Erzeugung des Objektmoduls die relative Adresse des Elements (Adresse bezüglich der Basisadresse 0 des Objektmoduls) in Form eines Zusatzes `a` gespeichert. Zur Laufzeit wird eine nichtsymbolische Relokation der Form

```
*f = a + b;
```

durchgeführt.

Unter ELF ist eine Relokationstabelle meist in einer eigenen Sektion enthalten. Die Namen dieser Sektionen bilden sich nach dem Muster `.rel`*name* bzw. `.rela`*name*. Dabei bezeichnet *name* den Namen der von der Relokation betroffenen Sektion. So wird z. B. mittels der Relokationssektion `.rela.text` der Inhalt der Sektion `.text` reloziert. Informationen im Section Header einer Relokationssektion verweisen auf die von den Relokationen betroffene Sektion und die zur Durchführung symbolischer Relokationen verwendete Symboltabelle.

## 4.9 Bewertung von ELF

Das Objektdateiformat beeinflußt in entscheidendem Maß die Arbeit des Laufzeit-Linkers. Es verkörpert eine Datenstruktur, deren wohldurchdachter Entwurf dazu beiträgt, den Algorithmus des dynamischen Linkens einfach und effektiv zu gestalten. Das ist von besonderer Bedeutung, da durch den Laufzeit-Linker die Performanz einer Anwendung beeinflußt wird. Das Objektdateiformat muß die Verwaltung aller Informationen erlauben, die für die verschiedenen Konzepte des dynamischen Linkens benötigt werden. Die Bewertung von ELF soll anhand der folgenden Kriterien durchgeführt werden:

1. **Darstellung von Abhängigkeiten zwischen Objektmodulen**
   Durch die Bezeichnung der von einem dynamisch gelinkten Programm oder einer geteilt benutzbaren Objektdatei benötigten (weiteren) geteilt benutzbaren Objektdateien erlaubt ELF die Definition von Abhängigkeitsgraphen. Benötigte geteilt benutzbare Objektdateien können mit Hilfe eines im benutzenden Objektmodul enthaltenen Suchpfads (Runpath) lokalisiert werden.

2. **Initialisierungs- und Terminierungscode**
   Unter ELF wird Initialisierungs- und Terminierungscode geteilt benutzbarer Objektdateien in Form eigens dafür existierender Sektionen verwaltet. Durch den Link-Editor wird sichergestellt, daß diese Sektionen ausführbare Funktionen darstellen. Das Dynamic-Segment gibt den Eintrittspunkt dieser Funktionen an (s. Abschn. 5.5, S. 43).

3. **Typsicherheit**
   Bei Symbolwerten wird unter ELF nur zwischen Datenobjekten und Funktionen unterschieden. Es existieren keine speziellen Datenstrukturen, um Informationen über den Typ eines Datenobjekts sowie des Rückgabewerts und der Argumente einer Funktion abzuspeichern. Dieser Mangel kann jedoch durch die Verwendung dekorierter Symbolnamen ausgeglichen werden.

4. **Versionsverwaltung**
   Unter ELF wird eine einfache Versionsverwaltung auf der Ebene kompletter geteilt benutzbarer Objektdateien verwirklicht (s. Abschn. 5.9 S. 51).

5. **Effektivität des Ladeprozesses**
   Eine ladbare ELF-Objektdatei (dynamisch gelinktes Programm, geteilt benutzbare Objektdatei) ist in Segmente strukturiert, die direkt in Speichersegmente überführt werden können (s. Abschn. 4.4 S. 28).

6. **Effektivität des Symbolzugriffs**
   Die Symbolsuche innerhalb einer ladbaren ELF-Objektdatei wird durch ein Hashing-Verfahren

beschleunigt (s. Abschn. 4.7, S. 32). Symboldefinitionen bezeichnen Symbolwerte durch relative Adressen, die durch einfache Addition der Basisadresse des Objektmoduls zur Laufzeit in absolute Adressen überführt werden. Das Objektdateiformat bietet keine Möglichkeit, Informationen darüber zu speichern, welches der benötigten geteilt benutzbaren Objektdateien ein referenziertes Symbol definiert. Bei einer Symbolsuche muß damit stets ein gesamter Abhängigkeitsgraph durchsucht werden.

7. **geteilte Benutzbarkeit**
   Die geteilte Benutzbarkeit auf ELF-basierten Systemen wird (implizit) durch das Abbilden von Objektdateien erreicht. Relokationen führen dabei zu privaten Modifikationen (s. Abschn. 3.4, S. 17), was die wahlfreie Positionierung von geteilt benutzbaren Objektdateien im Adreßraum eines Prozesses erlaubt. Die geteilte Benutzbarkeit von Objektdateien wird durch die Verwaltung positionsunabhängigen Codes (s. Abschn. 5.8, S. 45) verbessert.

8. **Flexibilität und Portierbarkeit**
   ELF-Objektdateien können um beliebige neue Sektionen und/oder Segmente erweitert werden. Das ELF-Objektdateiformat ist auf einer Vielzahl von Architekturen (z. B. AT&T WE32100, SPARC, Intel 80386, Intel i860, Motorola M68000 und Motorola M88000) einsetzbar. Architekturspezifische Informationen (z. B. Relokationsinformationen) können leicht an die zugrundeliegende Plattform angepaßt werden.

9. **Plattform-Unabhängigkeit**
   ELF-Objektdateien verfügen über plattformunabhängig interpretierbare Header-Informationen, (s. Abschn. 4.2, S. 26), die den Zugang zu allen weiteren Strukturinformationen ermöglichen. Dadurch ist der Einsatz von ELF in cross-compilation-Entwicklungsumgebungen denkbar.

Zusammenfassend kann festgestellt werden, daß sich das ELF-Objektdateiformat gut als Basis für das dynamische Linken eignet.

# Kapitel 5

# Dynamisches Linken auf der Basis von ELF

Im vorangegangenen Kapitel wurden mit dem Objektdateiformat die Datenstrukturen beschrieben, die dem dynamischen Linken auf ELF-basierten Systemen zugrunde liegen. In diesem Kapitel sollen die Mechanismen und Verfahren erläutert werden, die diese Datenstrukturen verwenden.

Es wird gezeigt, wie die in Kap. 3 (S. 15) vorgestellten Konzepte des dynamischen Linkens auf ELF-basierten Systemen umgesetzt werden.

## 5.1 Link-Editor

Die Analyse des Link-Editors gehört nicht zu den Schwerpunkten dieser Arbeit. Da dieses Werkzeug aber in entscheidender Weise das dynamische Linken vorbereitet, soll seine Funktion in dem Maß erklärt werden, wie es für das Verständnis des Laufzeit-Linkers nötig ist.

Der Link-Editor `ld(1)` ist ein Kommandozeilen-Programm, das meist als letztes Element der Entwicklungsumgebung von einem Compiler-Treiber (z. B. `cc(1)`) aufgerufen und durch Optionen gesteuert wird.

Durch den Link-Editor werden zwei Hauptfunktionsweisen unterschieden, von denen die Art seiner Ein- und Ausgabe abhängt:

- **statisches Linken**
  In diesem Modus werden vom Link-Editor relozierbare Objektdateien oder statisch gelinkte Programme erzeugt. Hierfür verarbeitet der Link-Editor Archive oder relozierbare Objektdateien als Eingabe (s. Abb. 5.1).

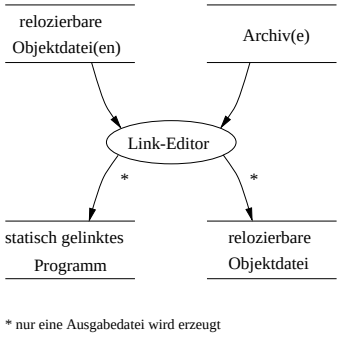


Abbildung 5.1: Link-Editor: statisches Linken

- **dynamisches Linken**
  In diesem Modus werden vom Link-Editor geteilt benutzbare Objektdateien oder dynamisch gelinkte Programme erzeugt. Der Link-Editor verarbeitet hierfür relozierbare oder geteilt benutzbare Objektdateien sowie Archive als Eingabe (s. Abb. 5.2).

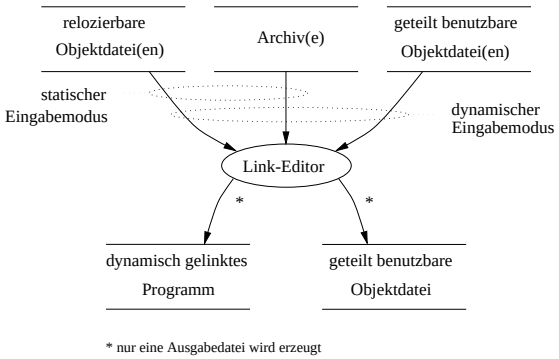


Abbildung 5.2: Link-Editor: dynamisches Linken

Beim dynamischen Linken kann durch Kommandozeilenoptionen der Eingabemodus spezifiziert werden. Der Eingabemodus legt fest, welche Objektdateitypen (bevorzugt) als Eingabe des Link-Editors verarbeitet werden sollen:

- **dynamischer Eingabemodus**
  Durch Angabe der entsprechenden Option[1] werden bevorzugt geteilt benutzbare Objektdateien verarbeitet, die Verwendung von relozierbaren Objektdateien und Archiven ist aber ebenfalls möglich. Die Angabe der Option `-l`*xxx* bewirkt dabei, daß der Link-Editor zuerst versucht, eine geteilt benutzbare Objektdatei `lib`*xxx*`.so` zu lesen. Findet er keine Datei diesen Namens, sucht er anschließend das Archiv `lib`*xxx*`.a`.
- **statischer Eingabemodus**
  Durch Angabe der entsprechenden Option[2] wird der Link-Editor angewiesen, ausschließlich relozierbare Objektdateien oder Archive als Eingabe zu verwenden.

Eingabemodus-Optionen können beim Aufruf des Link-Editors mehrfach angegeben werden und bestimmen den möglichen Typ für die jeweils nachfolgend spezifizierten Eingabedateien.

Die Funktion des Link-Editors gliedert sich in die Verarbeitung der Eingabedateien, die Symbolverarbeitung und die Erzeugung der Ausgabedateien:

1. **Verarbeitung der Eingabedateien**
   Der Link-Editor verarbeitet die Eingabedateien in der Reihenfolge ihrer Nennung als Parameter. Die in relozierbaren Objektdateien enthaltenen Sektionen mit Programminformationen (`.text`, `.data` usw.) werden dabei zu den gleichnamigen Sektionen der Ausgabedatei konkateniert. Sektionen mit Linkinformationen (`.strtab`, `rel.text` usw.) werden vom Link-Editor zum Aufbau interner Strukturen benutzt. So verwaltet der Link-Editor z. B. eine interne Symboltabelle, in der er definierte und undefinierte Symbole speichert (s. Symbolverarbeitung).

   Bei der Verarbeitung von Archiven wird ein darin enthaltenes Objektmodul (relozierbare Objektdatei) nur dann gelesen und in die Ausgabedatei kopiert, wenn es ein bis dahin undefiniertes Symbol definiert. Aus diesem Grund kann die Stellung eines Archivs in der `ld(1)`-Parameterliste von Bedeutung sein: Enthält ein Archiv ein Objektmodul $A$, das ein Symbol definiert, welches erst von einem später aufgeführten Objektmodul $B$ referenziert wird, so wird $A$ nicht gelesen.

[1] `-Bdynamic`, Standard
[2] `-Bstatic`

Als Faustregel sollte gelten, Archive als letztes in der `ld(1)`-Parameterliste zu spezifizieren. Benötigt ein verwendetes Modul des Archivs die Symboldefinition eines weiteren im Archiv enthaltenen Objektmoduls, so muß auch dieses Modul vom Link-Editor gelesen werden. Deshalb wird ein Archiv vom Link-Editor so oft durchmustert, bis ein Durchlauf auftritt, bei dem kein Objektmodul gelesen werden mußte.

Geteilt benutzbare Objektdateien werden verarbeitet, indem der Link-Editor ihren Namen in der Abhängigkeitsliste der Ausgabedatei vermerkt. Ihre Symboltabellen werden vom Link-Editor zur Prüfung der Auflösbarkeit symbolischer Referenzen der Ausgabedatei verwendet. Damit können Fehler, die sonst erst unter dem Laufzeit-Linker auftreten würden, aufgedeckt werden. Beim dynamischen Linken kann mittels einer Option des Link-Editors der Runpath (s. Abschn. 4.4.3, S. 30) in der Ausgabedatei gespeichert werden.

2. **Symbolverarbeitung**

   Der Link-Editor akkumuliert die Einträge der Symboltabellen relozierbarer Objektdateien in der internen Symboltabelle. Dopplungen von Symbolen gleichen Namens werden dabei mittels der Resolutionsregeln (s. Abschn. 2.6, S. 13) aufgelöst. Bei der Erzeugung eines Programms darf die interne Symboltabelle nach der Verarbeitung aller Eingabedateien keine undefinierten Symbole mehr enthalten. Im Fall der Erstellung relozierbarer oder geteilt benutzbarer Objektdateien stellen verbleibende undefinierte Symbole keinen Fehler dar.

3. **Erzeugung der Ausgabedatei**

   Nach der Verarbeitung der Eingabedateien und ihrer Symbole schreibt der Link-Editor die akkumulierten Sektionen mit Programm- und Linkinformationen in die Ausgabedatei. Die Adressen von Symbolwerten und Relokationsfeldern werden hierbei an deren Position in der Ausgabedatei angepaßt. Erzeugt der Link-Editor ein dynamisch gelinktes Programm oder eine geteilt benutzbare Objektdatei, so wird neben der „gewöhnlichen“ Symboltabelle (`.symtab`-Sektion) eine weitere Symboltabelle speziell für das dynamische Linken erstellt (`.dynsym`-Sektion, s. Abschn. 4.6, S. 32). Daneben generiert der Link-Editor analog zur `.strtab`-Sektion eine mit der `.dynsym`-Sektion korrespondierende Stringtabelle (`.dynstr`-Sektion, s. Abschn. 4.5, S. 31). Die Position von `.dynsym`- und `.dynstr`-Sektion wird vom Link-Editor so gewählt, daß beide Bestandteil des Textsegments sind und mit dem dynamisch gelinkten Programm oder der geteilt benutzbaren Objektdatei in den Speicher geladen werden. Der Link-Editor ist außerdem für die Erzeugung des Dynamic-Segments (s. Abschn. 4.4.3, S. 30) verantwortlich.

## 5.2 Laufzeit-Linker

Der Laufzeit-Linker ist das Kernstück des dynamischen Linkens unter ELF. Er wird zum Start dynamisch gelinkter Programme verwendet. Der Laufzeit-Linker exportiert darüber hinaus eine Reihe von Funktionen, die von einem Programm zur Laufzeit zum expliziten dynamischen Linken geteilt benutzbarer Objektdateien verwendet werden können.

ELF-Dateien bieten die Möglichkeit, den Namen eines sogenannten *Interpreters* anzugeben. Als Interpreter soll in diesem Zusammenhang ein Programm(modul) verstanden werden, unter dessen Steuerung eine ELF-Datei gestartet wird. Ein spezieller PHT-Eintrag (Program-Header-Typ `PT_INTERP`) verweist auf den Pfadnamen des Interpreters. Der `exec(2)`-Systemruf lädt und startet den Interpreter nach dem Laden der ELF-Datei. Im Fall dynamisch gelinkter Programme stellt der Laufzeit-Linker diesen Interpreter dar. Der Laufzeit-Linker selbst ist eine geteilt benutzbare Objektdatei[3], durch die beim Start eines dynamisch gelinkten Programms die folgenden Operationen durchgeführt werden:

1. **Selbstrelokation**

   Mit der Durchführung von Relokationen paßt sich der Laufzeit-Linker an seine Position innerhalb des Adreßraums des Prozesses an. Dabei werden symbolische Referenzen innerhalb des Laufzeit-Linkers aufgelöst.

[3] Auf Solaris 2.x heißt diese Datei `ld.so.`*n*, auf Linux `ld-linux.so.`*n*. Durch *n* wird die Versionsnummer des Laufzeit-Linkers gekennzeichnet.

2. **Laden benötigter geteilt benutzbarer Objektdateien**
   Alle im Abhängigkeitsgraphen des dynamisch gelinkten Programms enthaltenen geteilt benutzbaren Objektdateien werden geladen (s. Abschn. 5.4, S. 43).

3. **Relokation**
   Das dynamisch gelinkte Programm und die geteilt benutzbaren Objektdateien werden reloziert.

4. **Initialisierung und Terminierung globaler Objekte**
   Der Initialisierungscode der geteilt benutzbaren Objektdateien wird ausgeführt. Ihr Terminierungscode wird mittels `atexit(3C)` registriert (s. Abschn. 5.5, S. 43).

5. **Start der Anwendung**
   Die Steuerung wird an den Eintrittspunkt des dynamisch gelinkten Programms übergeben.

Zur Lokalisierung der geteilt benutzbaren Objektdateien durch den Laufzeit-Linker werden die durch die folgenden Informationen spezifizierten Verzeichnisse in der angegebenen Reihenfolge durchmustert:

1. **Umgebungsvariable**
   Die Umgebungsvariable `LD_LIBRARY_PATH` enthält eine Liste von durch Doppelpunkt getrennten Pfadnamen, die sowohl vom Link-Editor als auch vom Laufzeit-Linker auf der Suche nach Objektdateien durchmustert wird. Von dieser Möglichkeit sollte sparsam Gebrauch gemacht werden, da die globale Gültigkeit dieser Variablen für alle dynamisch gelinkten Programme eine erhebliche Vergrößerung des Suchraums des Laufzeit-Linkers mit sich bringt.

2. **Runpath**
   Hinter dem Runpath verbirgt sich ein Eintrag im Dynamic-Segment eines dynamisch gelinkten Programms oder einer geteilt benutzbaren Objektdatei, der auf eine Liste von durch Doppelpunkt getrennten Pfadnamen verweist. Der Runpath ist die empfohlene Variante zur Lokalisierung benötigter geteilt benutzbarer Objektdateien.

3. **Standardverzeichnis**
   Das Verzeichnis `/usr/lib` wird durchsucht, wenn eine Objektdatei nicht mittels der voranstehenden Möglichkeiten lokalisiert werden konnte.

Nach dem Laden der geteilt benutzbaren Objektdateien werden diese zusammen mit dem dynamisch gelinkten Programm vom Laufzeit-Linker reloziert. Zur Ermittlung der dafür benötigten Symboldefinitionen wird der Abhängigkeitsgraph mittels Breitensuche durchmustert. Dabei wird vom Laufzeit-Linker die *Interposition* als Resolutionsregel verwendet (s. Abschn. 2.6, S. 13).

Eine Ausnahme bei der Durchführung der Relokationen bilden die Einträge der Procedure Linkage Table (PLT). Diese werden vorläufig so reloziert, daß durch sie die Steuerung an den Laufzeit-Linker übertragen wird. Dieser führt ihre endgültige Relokation erst dann durch, wenn die dazugehörige Funktion aufgerufen wird (s. Abschn. 5.8.2, S. 49).

Nach der Übergabe der Steuerung an das Programm wird der Laufzeit-Linker erneut aktiv, wenn ein PLT-Eintrag reloziert werden muß. Darüber hinaus stellt der Laufzeit-Linker seine Funktionalität über die in `dlfcn.h` spezifizierte Programmierschnittstelle zur Verfügung. Mit den darin enthaltenen Routinen kann ein Programm selbst zur Laufzeit Objektdateien linken und auf die darin enthaltenen Symboldefinitionen zugreifen.

Das Verhalten des Laufzeit-Linkers wird durch die folgenden Umgebungsvariablen beeinflußt:

- `LD_PRELOAD`
  Durch `LD_PRELOAD` kann eine Liste von Namen geteilt benutzbarer Objektdateien angegeben werden, die vom Laufzeit-Linker unmittelbar nach einem dynamisch gelinkten Programm und noch vor den von ihm benötigten geteilt benutzbaren Objektdateien geladen werden. Die Symboldefinitionen der auf diese Weise spezifizierten Objektdateien genießen aufgrund der Interposition

Vorrang vor denen der „regulären“ benötigten geteilt benutzbaren Objektdateien. Damit ist eine einfache Möglichkeit der Änderung der Funktionalität eines dynamisch gelinkten Programms gegeben.

- `LD_BIND_NOW`
  Standardmäßig führt der Laufzeit-Linker die Relokation eines PLT-Eintrags erst in dem Augenblick aus, in dem die entsprechende Funktion zum ersten Mal aufgerufen wird (s. Abschn. 5.7, S. 45). Durch die Definition von `LD_BIND_NOW` wird der Laufzeit-Linker veranlaßt, PLT-Relokationen schon während des Programmstarts durchzuführen.

## 5.3 Laden einer ELF-Objektdatei

Der Inhalt einer geteilt benutzbaren Objektdatei wird segmentweise (s. Abschn. 4.4, S. 28) mittels des `mmap(2)`-Systemrufs in den Adreßraum eines Prozesses abgebildet. Ein PHT-Eintrag gibt die Position eines Segments innerhalb der Objektdatei an und spezifiziert außerdem durch eine relative Adresse die Hauptspeicherposition des Segments bezüglich der Basisadresse der Datei.

Speicherzugriffsrechte für den Inhalt einer ELF-Objektdatei werden auf der Ebene von Segmenten festgelegt. Die Hauptspeicherverwaltung hingegen vergibt Zugriffsrechte meist auf der Ebene von Speicherseiten. Aus diesem Grund darf die Grenze zweier aufeinanderfolgender Segmente mit unterschiedlichen Speicherzugriffsrechten nicht in eine Speicherseite fallen. Segmentgrenzen werden jedoch nicht auf Speicherseitengrenzen ausgerichtet. Vielmehr erzeugt der Link-Editor die relativen Adressen von Segmenten so, daß zwischen der letzten Adresse eines Segments und der ersten Adresse des Folgesegments ein Zwischenraum gemäß der maximalen Speicherseitengröße des zugrundeliegenden Systems[4] vorhanden ist. Bedingt durch den Abbildungsprozeß können deshalb in der ersten Seite eines Segments die letzten Daten des vorangegangenen Segments enthalten sein, während in der letzten Seite eines Segments die ersten Daten des nachfolgenden Segments enthalten sein können.

Zur Darstellung des Ladeprozesses wird folgende (vereinfachte) Funktion zum Einblenden des Inhalts einer Datei in den Adreßraum eines Prozesses verwendet:

```
map(a, s, fd, filepos)
```

Mit `fd` wird dabei die Datei bezeichnet, aus der beginnend ab Position `filepos` `s` Bytes an die Adresse `a` des Prozesses eingeblendet werden. Zur Vereinfachung der Darstellung werden bezüglich der Speicherseitengröße `PAGESIZE` die folgenden drei Funktionen `PPB`, `SPB` und `PBD` definiert. Mittels `PPB` (*P*receding *P*age *B*oundary) wird die größte Seitengrenze kleiner oder gleich der Adresse `a` ermittelt:

```
PPB(a) = a & ~(PAGESIZE - 1)
```

Durch `SPB` (*S*ucceeding *P*age *B*oundary) wird die kleinste Seitengrenze größer oder gleich der Adresse `a` berechnet:

```
SPB(a) = PPB( a + (PAGESIZE - 1))
```

Mittels `PBD` (*P*age *B*oundary *D*isplacement) wird der Abstand der Adresse `a` von der nächstkleineren Seitengrenze angegeben:

```
PBD(a) = a & (PAGESIZE - 1)
```

Abbildung 5.3 stellt das Segment $S_n$ einer geteilt benutzbaren Objektdatei dar, die an die Basisadresse `baddr` innerhalb eines Prozesses eingeblendet wird. Durch `raddr`, `filesz` und `memsz` werden die in der Objektdatei enthaltenen Angaben über die relative Startadresse sowie über die Datei- und Hauptspeichergröße des betrachteten Segments bezeichnet. Das Segment beginnt innerhalb der Objektdatei an der Position `offset`.

[4] Beispiele für die max. Seitengröße: SPARC → 8KB; Intel 80386, Intel i860 → 4KB

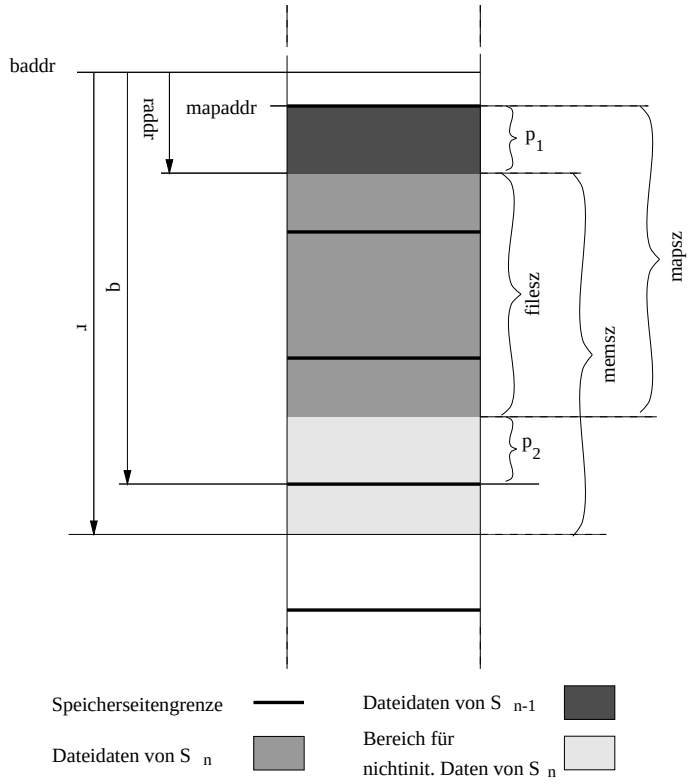


Abbildung 5.3: Laden eines Segments

Das Segment wird dabei gemäß folgender Anweisungen in den Adreßraum eingeblendet:

```
mappaddr = PPB( baddr + raddr );
mapsz    = filesz + p1
         = filesz + PBD( raddr );
filepos  = PPB( offset );

map( mapaddr, mapsz, fd, filepos );
```

Durch `p1` wird die Größe des Bereichs angegeben, der die Daten des Vorgänger-Segments $S_{n-1}$ enthält, durch `p2` wird die Größe des bis zur nächsten Seitengrenze verbleibenden Speicherplatzes bezeichnet, der vom Laufzeit-Linker mit Hinblick auf die Behandlung der `.bss`-Sektion mit 0 initialisiert wird. Besondere Beachtung gilt dem Ende des Datensegments, das ggf. die `.bss`-Sektion beinhaltet. Diese Sektion soll zur Laufzeit nichtinitialisierte Daten beherbergen, belegt jedoch keinen Platz in der Datei. Deshalb muß durch den Laufzeit-Linker Hauptspeicher für diese Sektion allokiert und mit 0 initialisiert werden. Zu diesem Zweck hat die Speichergröße des Datensegments (`memsz`) einen höheren Wert als die Dateigröße (`filesz`). Der Laufzeit-Linker muß für das Segment `memsz` Bytes allokieren, und die letzten `memsz` – `filesz` Bytes mit 0 initialisieren. Mit den zuvor angegebenen Anweisungen gestaltet sich dieser Schritt wie folgt:

```
q = SPB( raddr + filesz );
r = raddr + memsz;
if ( q < r ){
        mapaddr = baddr + q;
        mapsz   = r - q;
        map( mapaddr, mapsz, /dev/zero, 0 );
}
```

Der Speicherplatz wird hierbei durch Einblenden eines Abschnitts der speziellen Datei `/dev/zero` allokiert und auf diese Weise mit 0 initialisiert.

## 5.4 Abhängigkeitsgraph

Wird ein dynamisch gelinktes Programm oder eine geteilt benutzbare Objektdatei mittels einer (weiteren) geteilt benutzbaren Objektdatei $M$ erzeugt, so vermerkt der Link-Editor den Dateinamen von $M$ durch einen Eintrag im Dynamic-Segment (Typ des Eintrags: `DT_NEEDED`) der Ausgabedatei (s. Abschn. 4.4.3, S. 30). Alle `DT_NEEDED`-Einträge des Dynamic-Segments eines Moduls zusammen formen dessen *Abhängigkeitsliste*. Betrachtet man die Objektmodule einer dynamisch gelinkten Anwendung als Knoten und verbindet diese durch gerichtete Kanten mit den Elementen ihrer Abhängigkeitslisten, so ergibt sich der *Abhängigkeitsgraph*. Auf der Suche nach Symboldefinitionen durchmustert der Laufzeit-Linker mittels Breitensuche stets den gesamten Abhängigkeitsgraphen. Die Durchmusterung beginnt bei dem Objektmodul, das die Wurzel des Abhängigkeitsgraphen darstellt (z.B. beim dynamisch gelinkten Programm). Die Kantenmenge des Graphen kann dadurch reduziert werden (im Extremfall kann der Abhängigkeitsgraph auf einen Baum zurückgeführt werden). So braucht z. B. ein Objektmodul, welches Symboldefinitionen seines „Vaters“ im Abhängigkeitsgraphen benötigt, diesen nicht durch einen `DT_NEEDED`-Eintrag zu referenzieren. Der Abhängigkeitsgraph wird beginnend beim dynamisch gelinkten Programm rekursiv aufgebaut. Um das wiederholte Laden eines Objektmoduls zu vermeiden, verwaltet der Laufzeit-Linker eine Liste geladener Module. Diese wird vor dem Laden eines Objektmoduls durchsucht. Wird ein Objektmodul in dieser Liste gefunden, so wird die dadurch bezeichnete Hauptspeicherrepräsentation eines Objektmoduls benutzt.

## 5.5 Initialisierung und Terminierung globaler Objekte

Die Initialisierung und Terminierung globaler Objekte wird durch Initialisierungs- und Terminierungscode (s. Abschn. 3.7, S. 21) durchgeführt. Der in einer ausführbaren Datei enthaltene Initialisierungs- und Terminierungscode wird vom Programmcode der Datei selbst zur Ausführung gebracht. Der Initialisierungs- und Terminierungscode geteilt benutzbarer Objektdateien muß hingegen durch den Laufzeit-Linker behandelt werden.

ELF stellt für den Initialisierungs- und Terminierungscode eines Objektmoduls die speziellen Sektionen `.init` und `.fini` bereit. Im Fall geteilt benutzbarer Objektdateien wird der Inhalt dieser Sektionen durch die Entwicklungsumgebung zu parameterlosen Funktionen ohne Rückgabewert ergänzt. Diese Funktionen sollen im weiteren als *Initialisierungs-* und *Terminierungsfunktion* einer geteilt benutzbaren Objektdatei bezeichnet werden. Der Laufzeit-Linker verwendet spezielle Einträge im Dynamic-Segment der geteilt benutzbaren Objektdatei (Typ der Einträge: `DT_INIT` bzw. `DT_FINI`), um den Eintrittspunkt dieser Funktionen zu bestimmen. Die meisten Entwicklungsumgebungen erzeugen darüber hinaus die Symboldefinitionen `_init` und `_fini` für die Initialisierungs- und Terminierungsfunktion einer geteilt benutzbaren Objektdatei.

Die Initialisierungsfunktionen der im Abhängigkeitsgraphen einer dynamisch gelinkten Anwendung enthaltenen geteilt benutzbaren Objektdateien werden vom Laufzeit-Linker in umgekehrter Reihenfolge der Breite-zuerst-Durchmusterung (Breadth-First-Traversal, BFT) des Graphen ausgeführt. Das entspricht der in Abschn. 3.7 (S. 22) geforderten Reihenfolge für die Abarbeitung von Initialisierungscode.

Die Terminierungsfunktionen geteilt benutzbarer Objektdateien werden vom Laufzeit-Linker beim Laden mittels `atexit(3C)` in umgekehrter BFT-Reihenfolge registriert. Bei der Beendigung einer dynamisch gelinkten Anwendung wird durch den Programmcode des dynamisch gelinkten Programms zuerst der darin enthaltene Terminierungscode ausgeführt. Anschließend werden durch den Aufruf von `exit(3C)` alle mittels `atexit(3C)` registrierten Funktionen aufgerufen. Das geschieht in umgekehrter Registrierreihenfolge und damit in BFT-Reihenfolge. So wird die in Abschn. 3.7 (S. 22) geforderte Reihenfolge für die Abarbeitung von Terminierungscode erreicht.

In Abb. 5.4 ist ein Beispiel für die Abarbeitungsreihenfolge von Initialisierungs- und Terminierungscode dargestellt.

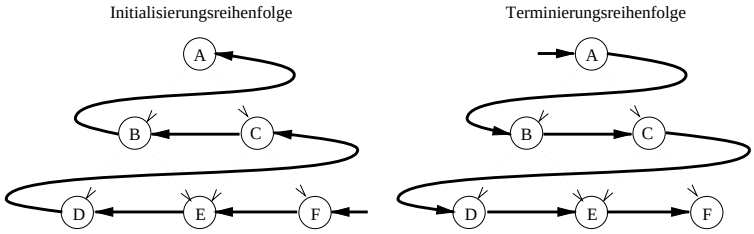


Abbildung 5.4: Abarbeitungsreihenfolge des Initialisierungs- und Terminierungscodes

Beim expliziten dynamischen Linken (s. Abschn. 3.1, S. 15) werden die Initialisierungsfunktionen der im Abhängigkeitsgraphen der zu linkenden geteilt benutzbaren Objektdatei enthaltenen Module vom Laufzeit-Linker ebenfalls in umgekehrter BFT-Reihenfolge aufgerufen. Die Terminierungsfunktionen müssen bei Beendigung der Nutzung des explizit gelinkten Objektmoduls in BFT-Reihenfolge ausgeführt werden.

Wenn der Initialisierungs- und Terminierungscode einer geteilt benutzbaren Objektdatei in Form einfacher Funktionen verarbeitet wird, so wirft das die Frage auf, warum zu diesem Zweck eigene Sektionen bereitgestellt werden. Der Grund dafür besteht in der Konkatenation gleichnamiger Sektionen von relozierbaren Objektdateien durch den Link-Editor. Folgendes Beispiel soll der Erläuterung dienen: Eine geteilt benutzbare Objektdatei `S` bestehe aus dem Inhalt der relozierbaren Objektdateien `A` und `B`, die beide über Initialisierungs- und Terminierungscode verfügen sollen. Wäre dieser Code in `A` und `B` in den Funktionen `init()` und `fini()` enthalten, müßten diese Funktionen in `S` zur Vermeidung von Konflikten eindeutige Namen (z. B. `init_A()` und `init_B()` bzw. `fini_A()` und `fini_B()`) erhalten. Anschließend wären in `S` neue Funktionen `init()` und `fini()` zu erzeugen, die ihrerseits `init_A` und `init_B` bzw. `fini_A()` und `fini_B` aufrufen. Dieser Weg wird nicht beschritten. Stattdessen ist der Initialisierungs- und Terminierungscode von `A` und `B` in den Sektionen `.init` und `.fini` beider Dateien enthalten, die vom Link-Editor zu den gleichnamigen Sektionen von `S` konkateniert werden. Anschließend sind diese Sektionen von `S` durch Voranstellen eines Funktionsprologs und Anhängen eines Funktionsepilogs zu einer Initialisierungs- bzw. Terminierungsfunktion zu ergänzen. Dies geschieht, indem durch die Entwicklungsumgebung ein Link-Editor-Aufruf generiert wird, bei dem die Liste der zu linkenden Objektdateien durch zwei spezielle Objektdateien – meist `crti.o` und `crtn.o` genannt – geklammert wird. Für das Beispiel würde der Link-Editor also in der Form

```
$ ld -o S crti.o A B crtn.o
```

aufgerufen. Durch `crti.o` werden `.init`- und `.fini`-Sektion mit einem Funktionsprolog eröffnet, durch `crtn.o` mit einem Funktionsepilog abgeschlossen. Der genaue Aufbau und Inhalt von Initialisierungs- und Terminierungscode variiert mit unterschiedlichen Entwicklungsumgebungen. Für konkrete Beispiele sei auf Anh. B verwiesen.

## 5.6 Geteilte Benutzbarkeit

Unter UNIX System V Release 4 kommt eine virtuelle Speicherverwaltung zum Einsatz, die das Abbilden von Dateien in den Hauptspeicher erlaubt (memory mapped files). Bilden mehrere Prozesse gleichzeitig eine Datei ab, so wird vom System dieselbe Hauptspeicherrepräsentation der Datei in den Adreßraum der Prozesse eingeblendet. Dadurch können abgebildete Dateien als gemeinsamer Speicher angesehen werden (s. [ATT91b]). Zum Abbilden von Dateien in den Hauptspeicher wird die `mmap(2)`-Routinenfamilie verwendet.

Der Laufzeit-Linker fügt die Segmente einer geteilt benutzbaren Objektdatei durch *private Abbildungen* in den Adreßraum eines Prozesses ein (das `MAP_PRIVATE`-Flag beim Aufruf von `mmap` ist

gesetzt). Bei privaten Abbildungen tritt die sogenannte *copy-on-write-policy* in Kraft. Sie bewirkt, daß jeder Prozeß, der eine Modifikation an einer Speicherseite der abgebildeten Datei vornimmt, eine lokale Kopie dieser Seite erhält. Für das Datensegment einer geteilt benutzbaren Objektdatei ist diese Vorgehensweise durchaus wünschenswert. Dadurch wird erreicht, das variable Datenobjekte, die zum Prozeßstatus gehören, mit der ersten Modifikation durch den Prozeß in dessen privat genutzten Speicherseiten enthalten sind.

Durch private Modifikationen des Textsegments wird es möglich, den Inhalt einer geteilt benutzbaren Objektdatei an eine beliebige Adresse innerhalb eines Prozesses einzublenden (s. Abschn. 3.4, S. 17). Dies bewirkt jedoch auch, daß effektiv weniger Programmcode-Speicherseiten geteilt benutzt werden können. Mit positionsunabhängigem Code (s. Abschn. 5.8, S. 45) wird eine Methode vorgestellt, mit der diesem Effektivitätsverlust entgegengewirkt werden kann.

## 5.7 Verzögertes Binden

Im Abschn. 3.8 (S. 22) wurde verzögertes Binden als Mechanismus beschrieben, bei dem die Auflösung symbolischer Referenzen erst dann erfolgt, wenn das referenzierte Datenobjekt oder die referenzierte Funktion tatsächlich benutzt (aufgerufen) wird.

Auf ELF-basierten Systemen können Funktionen verzögert gebunden werden. Zu diesem Zweck werden Funktionsaufrufe vom Übersetzungssystem so erzeugt, daß sie die Steuerung nicht direkt, sondern über eine Indirektionstabelle an die zu rufende Funktion übertragen. Die hierfür verwendete Indirektionstabelle wird als *Procedure Linkage Table* (PLT) bezeichnet. Die Einträge der PLT werden vom Laufzeit-Linker nach dem Laden eines Objektmoduls vorläufig so reloziert, daß bei deren erster Benutzung eine Routine des Laufzeit-Linkers gerufen wird. Diese Routine soll im folgenden als *Resolver* bezeichnet werden.

Der Resolver führt die endgültige Relokation eines PLT-Eintrags durch. Hierbei handelt es sich um eine symbolische Relokation, und beim Aufruf des Resolvers muß durch Angabe eines Parameters eine Relokationsinformation spezifiziert werden. Aus diesem Grund genügt es nicht, die PLT als einen Adreßvektor darzustellen, dessen Elemente als Argument indirekt adressierter Funktionsaufrufe verwendet werden. Stattdessen ist ein vorläufig relozierter PLT-Eintrag eine kurze Befehlssequenz, die den Resolver unter Angabe des oben genannten Parameters aufruft. Die Befehlssequenz des vorläufig relozierten PLT-Eintrags wird vom Resolver mittels der symbolischen Relokation durch eine Sprunganweisung zum Eintrittspunkt der zu rufenden Funktion überschrieben. Eine Sprunganweisung wird verwendet, damit die Rückkehradresse des ursprünglichen Funktionsaufrufs erhalten bleibt. Damit wird bei jeder weiteren Benutzung des PLT-Eintrags die Steuerung direkt an die zu rufende Funktion übertragen. Da der Resolver aufgrund eines Funktionsaufrufs aktiviert wurde, muß er abschließend die entsprechende Funktion zur Ausführung bringen.

Die PLT dient neben dem verzögerten Binden auch der Effektivitätssteigerung bei der geteilten Benutzung eines Objektmoduls durch positionsunabhängigen Code und wird unter diesem Aspekt im Abschn. 5.8.2 (S. 49) detaillierter beschrieben.

## 5.8 Positionsunabhängiger Code

Durch private (prozeßlokale) Modifikationen des Textsegments kann eine geteilt benutzbare Objektdatei auf beliebige Adressen innerhalb des Adreßraums der sie benutzenden Prozesse eingeblendet werden (s. Abschn. 3.4, S. 17). Ein Nachteil dieses Verfahrens ist, daß jede Modifikation des Textsegments durch eine Relokation zu einer privaten Kopie der Speicherseite führt, die das Relokationsfeld enthält. Durch diese Vorgehensweise geht der Vorteil des verringerten Hauptspeicherbedarfs aufgrund der Benutzung gemeinsamen Speichers verloren.

Aus diesem Grund ergibt sich ein neuer Aspekt bei der Betrachtung der geteilten Benutzbarkeit: Geteilte Benutzbarkeit kann nicht mehr nur als qualitative Eigenschaft, sondern auch als quantitative Eigenschaft aufgefaßt werden, die sich mit zunehmender Anzahl der am Textsegment durchzuführenden Relokationen verschlechtert.

Mit positionsunabhängigem Code (*Position Independent Code*, PIC) existiert ein Mechanismus, der es erlaubt, geteilt benutzbare Objektdateien an beliebigen Adressen innerhalb eines Prozesses einzublenden, ohne daß die geteilte Benutzbarkeit des Moduls darunter leidet. Das Grundprinzip des positionsunabhängigen Codes besteht darin, Relokationsfelder aus dem Text- in das Datensegment zu verschieben. Dadurch ist lediglich das Datensegment (das ohnehin nicht geteilt benutzt werden kann) von Relokationen betroffen. Die Relokationsfelder werden dabei in zwei Indirektionstabellen – die *Global Offset Table* (GOT) und die Procedure Linkage Table (PLT) – verlagert. Die folgenden beiden Abschnitte widmen sich der Verwendung dieser Tabellen.

Positionsunabhängiger Code wird vom Übersetzungssystem erzeugt. Der Laufzeit-Linker stellt im Zusammenhang mit PIC spezielle Funktionen zur Verfügung und führt eigens dafür generierte Relokationen aus.

### 5.8.1 Global Offset Table

Jede geteilt benutzbare Objektdatei, die auf globale Datenobjekte zugreift, erhält eine Global Offset Table. Die GOT wird mit der Modifikation durch spezielle Relokationen zum privaten Bestandteil des Prozesses, der die geteilt benutzbare Objektdatei verwendet. Mit Hilfe der Global Offset Table wird indirekt auf globale Datenobjekte zugegriffen. Unter Verwendung der GOT verändern sich Anweisungen der Form

> Zugriff auf Speicherzelle `daddr`

in

> Zugriff auf die durch den GOT-Eintrag `i` adressierte Speicherzelle.

Durch eine Relokation muß dafür gesorgt werden, daß der durch `i` indizierte GOT-Eintrag die Adresse `daddr` enthält. Die GOT ist ein Adreßvektor, der innerhalb einer ELF-Objektdatei in der Sektion `.got` abgespeichert wird. Das lokal gebundene Symbol `_GLOBAL_OFFSET_TABLE_` markiert den Beginn dieser Sektion. Es entspricht folgender Definition[5]:

```
static void *_GLOBAL_OFFSET_TABLE_[n];
```

Das Ende der GOT wird in ihrem ersten Eintrag festgehalten:

```
_GLOBAL_OFFSET_TABLE_[0]= &_GLOBAL_OFFSET_TABLE[n]
```

Eine geteilt benutzbare Objektdatei erhält einen GOT-Eintrag für jedes globale Datenobjekt, das sie referenziert (also nicht etwa für jedes globale Datenobjekt, das sie definiert). Wird kein globales Datenobjekt referenziert, so ist die GOT bis auf den ersten Eintrag leer. Positionsunabhängiger Code kann keine Annahmen über die Position der GOT im Adreßraum eines Prozesses machen. Vielmehr nutzt man den Umstand, daß der Inhalt einer geteilt benutzbaren Objektdatei als Ganzes in den Adreßraum des nutzenden Prozesses eingeblendet wird und damit relative Adressen innerhalb der geteilt benutzbaren Objektdatei stets ihre Gültigkeit behalten. In Abb. 5.5 wird die Verwendung der GOT dargestellt. Kernstück des Verfahrens ist eine Befehlssequenz, die zur Laufzeit ihre eigene, absolute Adresse `paddr` innerhalb des Prozesses bestimmt. Jede Funktion verfügt in ihrem Prolog über eine solche Befehlssequenz. Daneben wird vom Übersetzungssystem für jede Funktion der Abstand `disp` von der Befehlssequenz zur GOT im Programmcode vermerkt. Jede Funktion ermittelt die Adresse der GOT durch die Addition von `paddr` und `disp` und bestimmt mittels des GOT-Index `i` die Adresse des gewünschten Datenbereichs.

Zur Erläuterung des GOT-Zugriffs dient Code-Beispiel 5.1:

[5] Diese Definition dient nur der Erläuterung. Das Symbol `_GLOBAL_OFFSET_TABLE_` kann nicht aus einem Programm heraus referenziert werden.

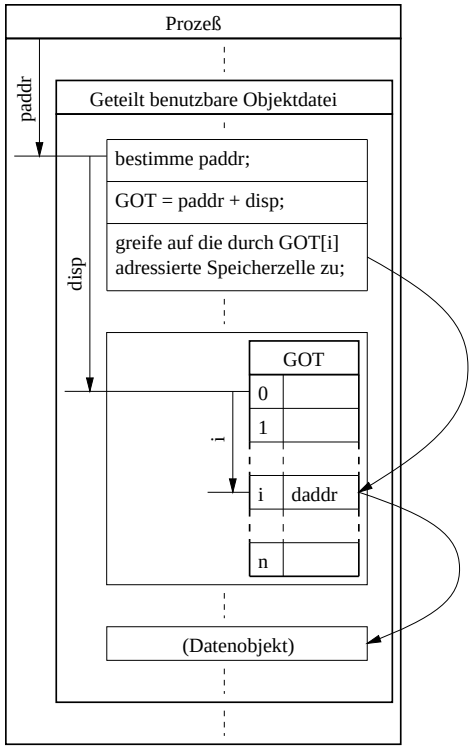


Abbildung 5.5: GOT-Zugriff

```
#include <stdio.h>

int global_int=42;

void global_func(){
  printf("global_int=%d\n", global_int);
}
```

**Code–Beispiel 5.1:** `libpic.c`

Das Code-Beispiel 5.2 stellt den mit der GOT in Zusammenhang stehenden (Pseudo-) Assembler-Code für das Code-Beispiel 5.1 dar. Durch den relativ adressierten `call`-Befehl in Zeile 02 wird die auf den Befehl folgende Adresse `x` angesprungen. Durch die Ausführung dieses Befehls (der im übrigen keinen Einfluß auf den Programmablauf hat), wird die dem `call`-Befehl folgende absolute Adresse als Rückkehr-Adresse verfügbar (z. B. auf dem Stack). Diese Adresse verkörpert – gemäß obiger Erläuterung – den Wert `paddr`.

Im Befehl in Zeile 04 wurde vom Übersetzungssystem mit `disp` der Abstand von `paddr` zur Global Offset Table als konstanter Wert festgehalten. Durch die Addition von `paddr` und `disp` wird die Adresse der GOT ermittelt. Nun wird in den Zeilen 05 und 06 mittels der GOT indirekt auf den String `_fmtstr`[6] (`"global_int=%d\n"`) und die Variable `global_int` zugegriffen. Die GOT selbst wird durch die Zeilen 13 – 15 repräsentiert. Vor dem Zugriff auf die GOT müssen deren Einträge durch den Laufzeit-Linker reloziert werden. Diese Relokationen werden durch die Informationen in den Zeilen 16 – 21 gesteuert. Die im (Pseudo-) Assembler-Code benutzte Notation

```
sym@tab
```

[6]Der Symbolname `_fmtstr` ist zur besseren Lesbarkeit gewählt worden. Das Übersetzungssystem generiert für Zeichenketten-Literale meist unbenannte Symbole.

bezeichnet den Eintrag der Indirektionstabelle `tab` für das Symbol `sym` (`global_int@GOT` bezeichnet also den GOT-Eintrag für die Variable `global_int`).

```
disassembly for libpic.so

section .text

global_func()
01:             ...
02:             call x
03:     x:      set paddr = <Rueckkehradresse des call-Befehls>
04:             set GOT = paddr + disp
05:             reg0    = [GOT + 4]          // _fmtstr
06:             reg1    = [GOT + 8]          // &global_int
07:             reg1    = [reg1]             // global_int
08:             call printf@PLT(reg0, reg1)
09:             ...
10:             return

section .rodata
11:             _fmtstr = "global_int=%d\n"

section .data
12:             global_int = 42

section .got
13:             GOT[0] = &GOT[3]            // GOT-Ende
14:             GOT[1] = _fmtstr            // GOT-Eintrag fuer _fmtstr
15:             GOT[2] = 0                  // GOT-Eintrag fuer global_int

section .rel.got

// GOT-Relokation 1
16:             Relokationsfeld   = &GOT[1] // &_fmtstr@GOT
17:             Relokationssymbol = 0       // nichtsymbolische Relokation
18:             Relokationstyp    = R_arch_RELATIVE

// GOT-Relokation 2
19:             Relokationsfeld   = &GOT[2] // &global_int@GOT
20:             Relokationssymbol = Symboltabellen-Index von global_int
21:             Relokationstyp    = R_arch_GLOB_DAT
```

**Code–Beispiel 5.2:** `libpic.so` (1)

Durch die GOT-Relokation 1 (Zeile 16 – 18) wird der GOT-Eintrag für das Zeichenketten-Literal `_fmtstr` reloziert. Das durch Zeile 14 repräsentierte Relokationsfeld enthält die relative Adresse von `_fmtstr`. Durch den Relokationstyp `R_arch_RELATIVE` wird zu dieser relativen Adresse die Basisadresse der geteilt benutzbaren Objektdatei innerhalb des Prozesses addiert. Durch die GOT-Relokation 2 (Zeile 19 – 21) wird der GOT-Eintrag für die Variable `global_int` reloziert. Durch den Relokationstyp `R_arch_GLOB_DAT`[7] wird angegeben, daß das durch Zeile 15 repräsentierte Relokationsfeld durch die absolute Adresse des Symbolwerts von `global_int` zu ersetzen ist.

Die Sektionen `.text`, `.rel.got` und `.rodata` werden durch den Link-Editor im Textsegment der geteilt benutzbaren Objektdatei angeordnet, während die GOT in Form der `.got`-Sektion zusammen mit der `.data`-Sektion in das Datensegment plaziert wird. Hierdurch erhöht sich die Effektivität bei der geteilten Benutzung des Textsegments.

[7] Die Bezeichnung `R_arch_RELATIVE` bzw. `R_arch_GLOB_DAT` orientiert sich an der ELF-üblichen Notation von Relokationstypen. Im konkreten Fall ist `arch` durch den Namen der zugrundeliegenden Architektur zu ersetzen.

### 5.8.2 Procedure Linkage Table

Im Abschn. 5.7 (S. 45) wurde das Prinzip der Verwendung der Procedure Linkage Table vorgestellt. In diesem Abschn. wird eine konkrete Darstellung des Verfahrens angegeben.

Jede geteilt benutzbare Objektdatei, die eine globale Funktion[8] benutzt, erhält dafür einen Eintrag in der Procedure Linkage Table. Lokale Funktionen werden – wenn es die zugrundeliegende Architektur erlaubt – durch relativ adressierte `call`-Befehle aufgerufen. Relativ adressierte `call`-Befehle innerhalb eines Objektmoduls sind stets positionsunabhängig, und daher wird in einem solchen Fall kein PLT-Eintrag generiert.

Die PLT wird innerhalb einer ELF-Objektdatei in der Sektion `.plt` abgespeichert. Ihre Einträge sind kurze Befehlssequenzen (sogenannte *Jump Slots*). Eine geteilt benutzbare Objektdatei enthält einen PLT-Eintrag für jede globale Funktion, die durch die geteilt benutzbare Objektdatei referenziert wird (nicht für jede globale Funktion, die die Objektdatei definiert). Verwendet eine geteilt benutzbare Objektdatei keine globale Funktion, gibt es innerhalb dieser Datei keine PLT (die Sektionen `.plt` und `.rel.plt` bzw. `.rela.plt` existieren nicht).

Ein PLT-Eintrag `i` zum Aufruf einer Funktion `f()` wird vom Übersetzungssystem so generiert, daß er in einem Übergabeparameter `REL` (z. B. ein Prozessor-Register) den Index eines Relokationstabelleneintrags speichert und anschließend durch eine Sprunganweisung die Steuerung an den speziellen PLT-Eintrag 0 übergibt. Der Relokationstabelleneintrag definiert die Adresse des PLT-Eintrags `i` als Relokationsfeld, das mit einer Sprunganweisung zum Eintrittspunkt der Funktion `f()` zu überschreiben ist.

Der spezielle PLT-Eintrag 0 wird vom Laufzeit-Linker so reloziert, daß er den Resolver aufruft. Dieser Eintrag bleibt während der gesamten Benutzung eines Objektmoduls unverändert. Der Resolver führt die durch `REL` bezeichnete Relokation aus und modifiziert damit den PLT-Eintrag `i` so, daß er von da an die Steuerung direkt an die zu rufende Funktion `f()` überträgt. Abbildung 5.6 stellt den Ablauf beim ersten Aufruf einer Funktion über die PLT dar.

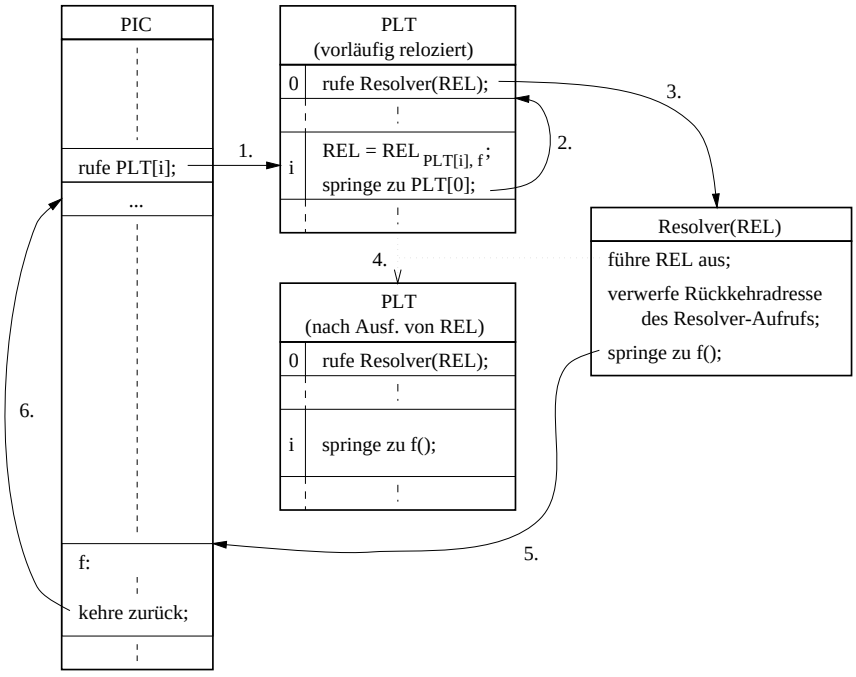


Abbildung 5.6: Initialer Aufruf eines PLT-Eintrags

Nachdem der Resolver einen PLT-Eintrag reloziert hat, wird bei allen weiteren Aufrufen der betreffenden Funktion die Steuerung durch den PLT-Eintrag direkt an die Funktion übertragen. Abbildung

[8] „Global“ bezeichnet in diesem Zusammenhang die globale Bindung des Funktionssymbols.

5.7 stellt diesen Vorgang dar.

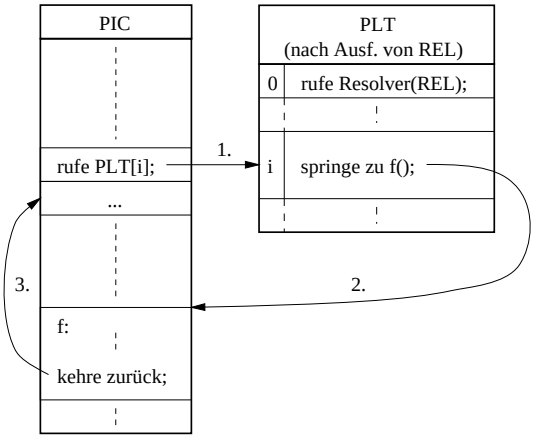


Abbildung 5.7: Nachfolgende Aufrufe eines PLT-Eintrags

Das Code-Beispiel 5.3 zeigt anhand von (Pseudo-) Assembler-Code den von der Entwicklungsumgebung im Zusammenhang mit der PLT erzeugten Inhalt der ELF-Objektdatei. Der in Zeile 26 – 27 dargestellte PLT-Eintrag `i` dient dem Aufruf der `printf`-Routine. Alle im Anweisungsteil der geteilt benutzbaren Objektdatei enthaltenen Aufrufe von `printf` verzweigen an diese Stelle. Der Übergabeparameter `REL` bezeichnet in der oben angegebenen Weise die Relokationsinformation, durch die in diesem Beispiel der PLT-Eintrag `i` mit einer Sprunganweisung zur Adresse von `printf` überschrieben wird. Mit diesem Übergabeparameter wird zum PLT-Eintrag 0 gesprungen.

```
section .plt

// PLT[0]:
22:              (nicht belegt)
23:

// PLT[1]:
24:              (nicht belegt)
25:
...

// PLT[i] (<--- call printf@PLT):
26:              REL = PLT-Relokation 1
27:              jump &PLT[0]

section .rela.plt

// PLT-Relokation 1:
28:              Relokationsfeld   = &PLT[i]      // printf@PLT
29:              Relokationssymbol = Symboltabellen-Index von printf
30:              Relokationstyp    = R_arch_JMP_SLOT
```

**Code–Beispiel 5.3:** `libpic.so` (2)

Der PLT-Eintrag 0, der bei der Erzeugung der ELF-Objektdatei nicht belegt wurde, muß beim Laden der Datei durch den Laufzeit-Linker mit einen Befehl zum Aufruf des Resolvers überschrieben werden. Der PLT-Eintrag 1 wird vom Laufzeit-Linker beim Laden der geteilt benutzbaren Objektdatei mit der Adresse einer Datenstruktur belegt, die dem Resolver alle für die Durchführung der PLT-Relokation nötigen Informationen bereitstellt. Dazu gehören

- die PLT-Relokationstabelle des betrachteten geteilt benutzbaren Objektmoduls und

- die Symboltabellen und Basisadressen der im Rahmen der dynamisch gelinkten Anwendung verwendeteten Objektmodule.

Code-Beispiel 5.4 zeigt, wie der Laufzeit-Linker die PLT-Einträge 0 und 1 initialisiert und wie PLT-Eintrag `i` durch die Ausführung der zugehörigen Relokation verändert wurde.

```
section .plt
// PLT[0]:

21:             ...
22:             call resolver

// PLT[1]:

23:             .word <Zeiger auf Link-Informationen>
24:
...

// PLT[i] (<--- call printf@PLT):

25:             jump printf
26:
```

**Code–Beispiel 5.4:** `libpic.so` (3)

Wie bei Verwendung der GOT kann auch hier festgestellt werden, daß durch die PLT Relokationsfelder aus dem Textsegment einer geteilt benutzbaren Objektdatei in deren Datensegment ausgelagert werden, wodurch sich die Effektivität bei der geteilten Benutzung des Textsegments verbessert. Können alle Referenzen zu globalen Funktionen und Datenobjekten indirekt durch PLT und GOT ausgedrückt werden, so ist das Textsegment einer geteilt benutzbaren Objektdatei uneingeschränkt geteilt benutzbar (es brauchen keine prozeßlokal verwendeten Kopien der Speicherseiten des Textsegments erzeugt werden).

## 5.9 Versionsverwaltung

Die Versionsverwaltung auf ELF-basierten Systemen beruht auf der Speicherung mehrerer Versionen einer geteilt benutzbaren Objektdatei unter einem *versionsbehafteten Dateinamen*. Ein versionsbehafteter Dateiname bildet sich nach folgendem Muster:

`lib`*name*`.so.`*maj*`.`*min*

Dabei bezeichnet *maj* die sogenannte *Major-Versionsnummer* und *min* die sogenannte *Minor-Versionsnummer*. Durch die Major-Versionsnummer wird die Kompatibilität mit anderen Objektmodulen und durch die Minor-Versionsnummer die Aktualität der geteilt benutzbaren Objektdatei angezeigt (s. Abschn. 3.6, S. 21). Zusätzlich enthält eine geteilt benutzbare Objektdatei einen Eintrag im Dynamic-Segment (Typ des Eintrags: `DT_SONAME`), der den versionsbehafteten Dateinamen *ohne* Minor-Versionsnummer angibt (s. Abb. 5.8). Unter diesem Namen wird ebenfalls ein Dateisystem-Link (Link 1) auf die neueste Version einer geteilt benutzbaren Objektdatei mit entsprechender Major-Versionsnummer verwaltet. Für die neueste Version einer geteilt benutzbaren Objektdatei (höchste Major- und Minor-Versionsnummer) wird unter einem *versionslosen Dateinamen* ein weiterer Dateisystem-Link (Link 2) eingerichtet. Ein versionsloser Dateiname bildet sich nach folgendem Muster:

`lib`*name*`.so`

Über Link 2 wird vom Link-Editor die neueste Version einer geteilt benutzbaren Objektdatei gefunden und zur Erstellung einer Ausgabedatei verwendet (Schritte 1. und 2. in Abb. 5.8). Der Link-Editor

vermerkt den `DT_SONAME`-Namen der verwendeten geteilt benutzbaren Objektdatei in der Abhängigkeitsliste der Ausgabedatei (Schritt 3. in Abb. 5.8). Bei der Verarbeitung dieser Ausgabedatei durch den Laufzeit-Linker wird über Link 1 die jeweils neueste kompatible Version einer benötigten geteilt benutzbaren Objektdatei gefunden und verwendet (Schritte 4., 5. und 6. in Abb. 5.8).

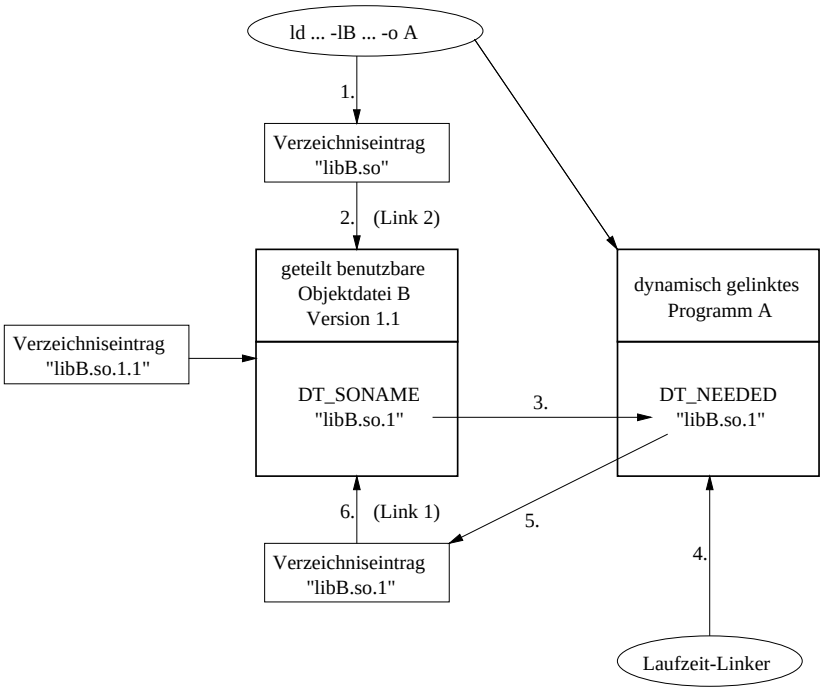


Abbildung 5.8: Versionsverwaltung unter ELF

Der `DT_SONAME`-Name einer geteilt benutzbaren Objektdatei wird bei deren Erstellung durch den Link-Editor festgelegt. Hierfür muß eine entsprechende Option beim Aufruf des Link-Editors angegeben werden. Die aufgeführten Maßnahmen zur Versionsverwaltung verlangen die aufmerksame Pflege von Dateisystem-Links und den richtigen Umgang mit Link-Editor-Optionen. Unter diesen Voraussetzungen kann die Kompatibilität von Objektmodulen zur Laufzeit gesichert und der Forderung nach Aktualität sowohl bei der Erstellung von Objektmodulen durch den Link-Editor als auch bei deren Verwendung durch den Laufzeit-Linker nachgekommen werden. Für den Laufzeit-Linker reduziert sich hierdurch die Versionsverwaltung auf die Prüfung der Übereinstimmung von Dateinamen.

# Kapitel 6

# Implementation eines dynamischen Linkers

Die Implementation des Prototyps eines dynamischen Linkers ist Bestandteil der Aufgabenstellung der vorliegenden Arbeit. Die Erklärungen in diesem Kapitel sollen dem Leser das Verständnis des Entwurfs, der Implementation und der Benutzung des Prototyps ermöglichen.

Der dynamische Linker verarbeitet geteilt benutzbare ELF-Objektdateien als Eingabe. Er wurde mit Hilfe der Programmiersprache C++ objektorientiert entwickelt. Implementationsplattform ist Linux, wobei zur Programmgenerierung der GNU-C++-Compiler[1] eingesetzt wurde. Der Entwurf des dynamischen Linkers stützt sich dabei an einigen Stellen auf Details aus [Engel95].

Im Zug der in diesem Kap. erläuterten Verfahren und Datenstrukturen wird an einigen Stellen auf Einzelheiten der Programmierung eingegangen. Es empfiehlt sich daher, die bezeichneten Quelltextbereiche nachzuschlagen.

## 6.1 Vorbetrachtungen

Im folgenden Abschn. werden Entwurfsaspekte bei der Implementation des dynamischen Linkers diskutiert. Hierbei wird vor allem auf Konzepte eingegangen, durch die sich die vorliegende Implementation von anderen Werkzeugen zum dynamischen Linken ([Ho91], [Engel95]) unterscheidet.

### 6.1.1 Globale Symbolmenge versus Gültigkeitsbereiche

Zur Auflösung symbolischer Referenzen eines Objektmoduls ist es i. allg. notwendig, auf die Symboldefinitionen anderer Objektmodule zuzugreifen. Besteht keine Möglichkeit, diese Nutzungsbeziehungen zwischen Objektmodulen im voraus festzulegen, müssen die Symboldefinitionen aller dynamisch gelinkten Objektmodule eine *globale Symbolmenge* $S$ formen, die zur Auflösung symbolischer Referenzen jedes Objektmoduls Verwendung findet. Bei dieser Vorgehensweise müssen die Symbolnamen aller Objektmodule eindeutig sein. Die Menge $S$ wird mit dem Laden jedes weiteren Objektmoduls um dessen Symboldefinitionen erweitert. Das hat zur Folge, daß nach dem Laden eines Objektmoduls $M_i$, das ein Symbol $s$ definiert, alle zuvor geladenen Objektmodule $M_j$ $(i \neq j)$ durchmustert werden müssen, um deren Referenzen zu $s$ aufzulösen. Diese Vorgehensweise wird mit zunehmender Anzahl geladener Objektmodule aufwendiger.

Im Fall des Objektdateiformats ELF besteht die Möglichkeit, Nutzungsbeziehungen zwischen Objektmodulen durch Abhängigkeiten darzustellen. Daraus ergeben sich Gruppen von Objektmodulen in Form von Abhängigkeitsgraphen, deren Symboldefinitionen als separate Symbolmengen aufgefaßt werden können. Der Programmierer hat dabei die Möglichkeit, Objektmodule so zu gruppieren, daß

[1] gcc Version 2.7.2

alle symbolischen Referenzen innerhalb eines Abhängigkeitsgraphen ausschließlich durch die darin enthaltenen Symboldefinitionen aufgelöst werden können[2]. Diese Eigenschaft eines Abhängigkeitsgraphen soll als *Abgeschlossenheit* bezeichnet werden. Aufgrund der Abgeschlossenheit braucht eine Symboldefinition nur noch innerhalb eines Abhängigkeitsgraphen sichtbar zu sein. Aus diesem Grund soll eine solche Gruppe von Objektmodulen auch als *Gültigkeitsbereich* bezeichnet werden. Unter dieser Voraussetzung wird es – im Gegensatz zur globalen Symbolmenge – möglich, die Relokation eines Objektmoduls als einmaligen Vorgang innerhalb eines Abhängigkeitsgraphen zu realisieren. Dabei reduziert sich außerdem der Suchraum bei der Lokalisierung von Symboldefinitionen, was eine Steigerung der Effektivität zur Folge hat. Weiterhin können in verschiedenen Gültigkeitsbereichen Symbole gleichen Namens definiert werden, wobei sichergestellt ist, daß in jedem Gültigkeitsbereich die „richtige“ Symboldefinition Verwendung findet.

Aufgrund dieser Vorteile wird die Menge der Objektmodule vom dynamischen Linker in der vorliegenden Implementation in Gültigkeitsbereiche strukturiert. Darüber hinaus besteht die Möglichkeit, zur Auflösung symbolischer Referenzen eines Abhängigkeitsgraphen $G_i$ Symboldefinitionen eines bezüglich der enthaltenen Objektmodule disjunkten Abhängigkeitsgraphen $G_j$ zu benutzen. Dieser als *globale Symbolsuche* bezeichnete Mechanismus bildet die Vorgehensweise der globalen Symbolmenge nach.

### 6.1.2 Abhängigkeitsgraph

Durch den dynamischen Linker werden Gültigkeitsbereiche in Form von Abhängigkeitsgraphen realisiert. Ein Abhängigkeitsgraph ist ein gerichteter azyklischer Graph, dessen Knoten Objektmodulen entsprechen und dessen Kanten durch die Abhängigkeiten der Objektmodule gebildet werden (s. Abb. 6.1).

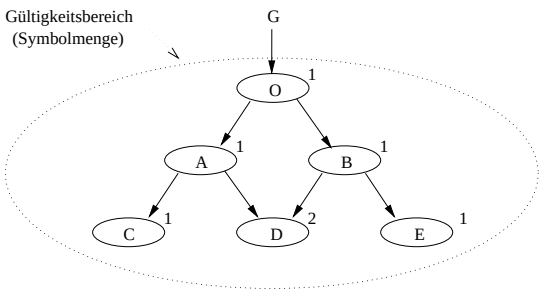


Abbildung 6.1: Abhängigkeitsgraph mit Referenzzählern

Die Symboldefinitionen aller in einem Abhängigkeitsgraphen $G$ enthaltenen Objektmodule formen zusammen die Symbolmenge $S_G$ von $G$. Eine symbolische Relokation eines solchen Objektmoduls wird dabei mittels einer Symboldefinition $s \in S_G$ durchgeführt. Das bedeutet z. B., daß das in Abb. 6.1 dargestellte Objektmodul $O$ Symboldefinitionen des Objektmoduls $D$ benutzen kann und umgekehrt. Auf diese Weise formt ein Abhängigkeitsgraph bezüglich der Symbolsuche den eingangs erwähnten Gültigkeitsbereich.

Definieren zwei Objektmodule innerhalb eines Gültigkeitsbereichs das gleiche Symbol $s$, so wird dieser Konflikt durch den Algorithmus zur Symbolsuche ausgeglichen, der in wohldefinierter Weise nur eine der beiden Symboldefinitionen (und zwar stets dieselbe) verfügbar macht. Die Vereinigung der Symboldefinitionen aller Objektmodule eines Abhängigkeitsgraphen $G$ zur Menge $S_G$ wird also nicht „physisch“ vollzogen, sondern ergibt sich vielmehr durch den Symbolsuche-Algorithmus. Für jeden Knoten des Abhängigkeitsgraphen wird ein Referenzzähler geführt, der angibt, wie viele Verweise auf diesen Knoten existieren.

[2] Eine Ausnahme bilden hierbei Referenzen zu Symbolen, die durch das zugrundeliegende System definiert werden, s. Abschn. 6.1.8, S. 60.

Ist ein Objektmodul $M_i$ Element mehrerer Abhängigkeitsgraphen $G_1, G_2, \ldots$, so wird es nicht mehrfach in den Hauptspeicher geladen. Vielmehr referenzieren $G_1, G_2 \ldots$ dieselbe Hauptspeicherrepräsentation des Objektmoduls. Dadurch sind die Symboldefinitionen aus $M_i$ und aller seiner Abhängigkeiten in allen Symbolmengen $S_{G_1}, S_{G_2}, \ldots$ enthalten (s. Abb. 6.2).

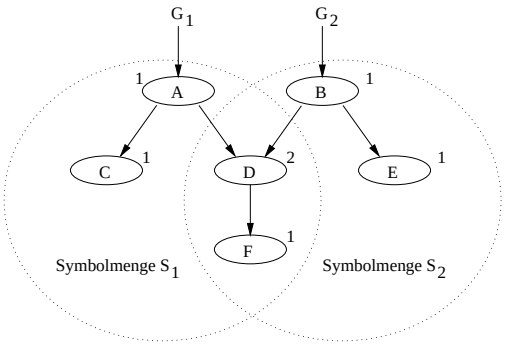


Abbildung 6.2: Objektmodule als Elemente mehrerer Abhängigkeitsgraphen

Dabei muß beachtet werden, daß $M_i$ nur einmal reloziert und initialisiert werden darf. Dies geschieht sinnfälliger Weise innerhalb des Gültigkeitsbereichs, der zuerst aufgebaut wird.

### 6.1.3 Funktionalität

Der implementierte Prototyp des dynamischen Linkers ermöglicht durch das Bereitstellen einer Programmierschnittstelle *explizites dynamisches Linken* (s. Abschn. 3.1, S. 15). Dabei werden geteilt benutzbare Objektdateien als Eingabe verarbeitet. Hierfür wird die durch die folgenden Operationen beschriebene Funktionalität realisiert:

- $G = link(n)$
  Mittels dieser Operation wird durch das Laden einer Objektdatei oder einer Gruppe von Objektdateien ein Abhängigkeitsgraph $G$ im Hauptspeicher aufgebaut. Dabei muß der Name $n$ der Objektdatei, die die Wurzel des Abhängigkeitsgraphen darstellt, als Parameter angegeben werden. Enthält der Hauptspeicher bereits ein angefordertes Objektmodul, wird dieses verwendet, anstatt die entsprechende Objektdatei erneut zu laden. Die *link*-Operation reloziert alle geladenen Objektmodule und initialisiert deren Datenbereiche.

- $a = getsym(G, s)$
  Durch diese Operation wird die Hauptspeicheradresse $a$ eines innerhalb des Abhängigkeitsgraphen $G$ definierten Symbols $s$ verfügbar gemacht.

- $unlink(G)$
  Mittels dieser Operation werden die Referenzzähler aller in einem Abhängigkeitsgraphen $G$ enthaltenen Objektmodule dekrementiert. Anschließend werden $G$ und alle darin enthaltenen, nicht mehr benötigten Objektmodule aus dem Hauptspeicher entfernt. Ein Objektmodul wird nicht mehr benötigt, wenn sein Referenzzähler den Wert 0 angenommen hat.

- $relink(G)$
  Durch die *relink*-Operation werden die innerhalb eines Abhängigkeitsgraphen $G$ im Hauptspeicher enthaltenen Objektmodule gemäß der entsprechenden Objektdateien aktualisiert (s. Abschn. 6.1.4, S. 56).

Bei der Durchführung der genannten Operationen werden vom dynamischen Linker die folgenden Konzepte berücksichtigt:

- **Versionsverwaltung**
  Der dynamische Linker realisiert eine einfache Versionsverwaltung auf der Basis einer Major- und Minor-Versionsnummer eines Objektmoduls (s. Abschn. 6.1.6, S. 58).

- **Initialisierungs- und Terminierungscode**
  Enthält ein Objektmodul Initialisierungs- bzw. Terminierungscode, so wird dieser nach dem Laden und Relozieren bzw. vor dem Löschen des Objektmoduls ausgeführt. Damit wird Initialisierungscode gewöhnlicherweise während einer *link*-Operation und Terminierungscode während einer *unlink*-Operation abgearbeitet. Wird durch eine *link*- oder *relink*-Operation ein Objektmodul durch eine neuere Version ersetzt, so wird der Terminierungscode der älteren Version abgearbeitet bevor der Initialisierungscode der neueren Version zur Ausführung gelangt.

- **Typsicherheit**
  Durch den dynamischen Linker wird Typsicherheit durch die Übereinstimmung dekorierter Symbolnamen bzw. Prototypnamen hergestellt (s. Abschn. 6.1.7, S. 59).

### 6.1.4 Relinken von Objektmodulen

Durch die *relink*-Operation werden die in einem Abhängigkeitsgraphen enthaltenen Objektmodule erneut aus den entsprechenden Dateien in den Hauptspeicher geladen. Bei der Durchführung der *relink*-Operation müssen folgende Konzepte Berücksichtigung finden:

- **Konsistenzwahrung**
  Es ist vorstellbar, die *relink*-Operation auf einfache Weise durch das Löschen aller in einem Abhängigkeitsgraphen $G_i$ enthaltenen Objektmodule und den anschließenden Neuaufbau von $G_i$ zu realisieren. Wird jedoch nach dem Löschen und noch vor dem Neuaufbau versucht, auf eine Symboldefinition aus $G_i$ zuzugreifen (z. B. über Referenzen aus anderen Abhängigkeitsgraphen $G_j$ $(i \neq j)$), so führt das zu einem undefinierten Zustand, der sich nur vermeiden läßt, wenn *relink* als unteilbare Operation implementiert wird. Dies erscheint jedoch mit Hinblick auf die Performanz als wenig effektive Lösung. Aus diesem Grund empfiehlt es sich, zuerst eine neue Version $G_i{}'$ des Abhängigkeitsgraphen $G_i$ aufzubauen und anschließend $G_i$ durch $G_i{}'$ zu ersetzen (s. Abb. 6.3). Dabei muß beachtet werden, daß andere Abhängigkeitsgraphen $G_j$ $(i \neq j)$ in $G_i$ enthaltene Objektmodule referenzieren können. Diese Referenzen müssen so umgewandelt werden, daß sie auf die neuen, in $G_i{}'$ enthaltenen Objektmodule verweisen.

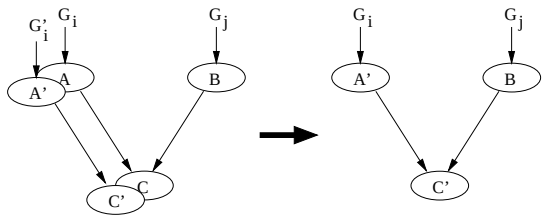


Abbildung 6.3: Einfache *relink*-Operation

- **Strukturvarianz**
  Beim Neuaufbau des Abhängigkeitsgraphen $G_i{}'$ kann es vorkommen, daß eine Referenz zu einem in $G_i$ enthaltenen Objektmodul $C$ entfällt. Ist $C$ gleichzeitig Element eines anderen Abhängigkeitsgraphen $G_j$ $(i \neq j)$, so muß sichergestellt werden, daß $C$ als Element von $G_j$ erhalten bleibt. Als Beispiel soll hier die in Abb. 6.4 dargestellte Konstellation dienen.

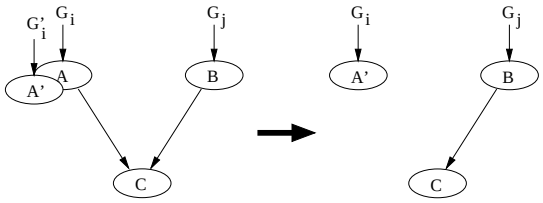


Abbildung 6.4: Strukturverändernde *relink*-Operation (1)

Weiterhin ist es möglich, daß ein in $G_i$ und $G_j$ enthaltenes Objektmodul $C$ in der aktualisierten Fassung ein Objektmodul $D$ referenziert. Hierbei muß dafür Sorge getragen werden, daß $D$ sowohl zum Bestandteil der neuen Fassung von $G_i$ als auch von $G_j$ wird. Abbildung 6.5 stellt ein Beispiel hierfür dar.

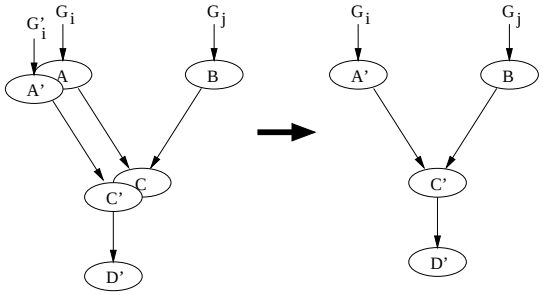


Abbildung 6.5: Strukturverändernde *relink*-Operation (2)

- **Granularität**

  Bisher wurde davon ausgegangen, daß mit einer *relink*-Operation stets ein gesamter Abhängigkeitsgraph aktualisiert wird. Das Relinken eines Blatts oder eines Subgraphen eines Abhängigkeitsgraphen $G_i$ kann dabei erreicht werden, indem mit der *link*-Operation ein neuer Identifikator $G_j$ für den entsprechenden Abschnitt des Abhängigkeitsgraphen geschaffen und anschließend die Operation

  $$relink(G_j)$$

  ausgeführt wird. Die Semantik der *link*-Operation bewirkt, daß die in $G_j$ enthaltenen Objektmodule nicht tatsächlich neu gelinkt, sondern nur ein weiteres Mal referenziert werden.

  Soll jedoch der dynamische Linker in Systemen zum Einsatz kommen, die sich durch eine feingranulare Adaptierbarkeit auszeichnen (s. [Schubert96]), so gewinnt die Möglichkeit an Bedeutung, Objektmodule separat austauschen zu können. Als Beispiel sei hier auf eine Klassenhierarchie verwiesen, die sich auf die in Abb. 6.6 dargestellte Weise in eine Objektdateihierarchie und damit in einen Abhängigkeitsgraphen abbilden läßt.

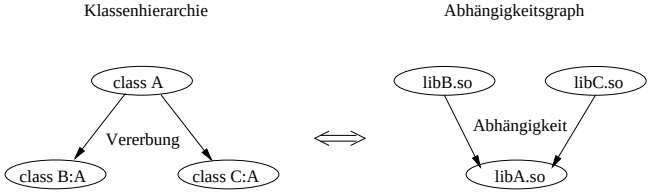


Abbildung 6.6: Klassenhierarchie und Abhängigkeitsgraph

Ändert sich die Implementation der abgeleiteten Klasse `B`, so muß mittels einer *relink*-Operation das Objektmodul `libB.so` ausgetauscht werden. Das Objektmodul `libA.so`, das die Basisklasse

`A` repräsentiert, bleibt von diesen Änderungen unberührt, und braucht daher nicht ausgetauscht zu werden. Diese Vorgehensweise kann realisiert werden, indem der *relink*-Operation durch einen zusätzlichen Parameter $m$ angezeigt wird, ob jeweils ein gesamter Abhängigkeitsgraph $G$ oder nur das Objektmodul, das die Wurzel von $G$ darstellt, von der *relink*-Operation betroffen sein soll. Die Syntax der *relink*-Operation nimmt damit die Form

$$relink(G, m)$$

an.

### 6.1.5 Re-Relokation

Im Zug einer *link*- oder *relink*-Operation kann eine bereits im Hauptspeicher enthaltene Version eines Objektmoduls $M$ von einer neu geladenen Version $M'$ ersetzt werden (s. Abschn. 6.1.6, S. 58). Hierbei ist zu beachten, daß alle Referenzen zu Symbolen aus $M$ ungültig werden, und damit jedes Objektmodul, das Symboldefinitionen aus $M$ benutzt, neu zu relozieren ist. Dieser Vorgang der erneuten Relokation soll als *Re-Relokation* bezeichnet werden.

Ein Ansatz zur Durchführung der Re-Relokation besteht darin, diejenigen Abhängigkeitsgraphen zu re-relozieren, in denen $M$ durch $M'$ ersetzt wurde. Hierbei ergibt sich das Problem, daß damit u. U. Objektmodule erneut reloziert würden, die keine Symboldefinitionen aus $M$ benutzt haben. So könnte z. B. nur Objektmodul $B$ in Abb. 6.2 (S. 55) von Symboldefinitionen aus $D$ Gebrauch machen, nicht hingegen Objektmodul $E$. Würde $D$ durch eine neue Version $D'$ ersetzt (z. B. durch eine *relink*-Operation über dem Graphen $G_1$), so würde $E$ vollkommen unnötig re-reloziert. Bedenkt man, daß Objektmodule möglicherweise zahlreiche Relokationen notwendig machen[3], so erscheint diese Vorgehensweise wenig effektiv.

Eine Lösung dieses Problems besteht darin, daß für jedes Objektmodul $M_i$ in einer speziellen Datenstruktur vermerkt wird, welche Objektmodule $M_j$ $(i \neq j)$ Symboldefinitionen aus $M_i$ benutzen. Diese Datenstruktur soll als *Liste effektiv nutzender Objektmodule* von $M_i$ bezeichnet werden. Wird das Objektmodul $M_i$ im Zug einer Ersetzung durch eine neue Version $M_i'$ gelöscht, so werden alle in dieser Liste enthaltenen Objektmodule $M_j$ markiert. Im anschließenden Prozeß der Re-Relokation über allen Abhängigkeitsgraphen werden die so markierten Objektmodule erneut reloziert.

Wird ein Objektmodul $M_j$ gelöscht, so muß auch der entsprechende Eintrag aus der Liste effektiv nutzender Objektmodule von $M_i$ entfernt werden (im oben aufgeführten Beispiel bedeutet das, daß mit dem Löschen von $G_2$ das Objektmodul $B$ aus der Liste effektiv nutzender Objektmodule von $D$ zu löschen ist). Aus diesem Grund wird für $M_j$ in einer weiteren Datenstruktur – der *Liste effektiv genutzter Objektmodule* von $M_j$ – ein Verweis auf $M_i$ vermerkt. Die Benutzung einer Symboldefinition eines anderen Objektmoduls führt also zu einer wechselseitigen Registrierung von Nutzer und Bereitsteller.

Bei der Re-Relokation muß weiterhin beachtet werden, daß ein durch die Adresse `relfld` bezeichnetes Relokationsfeld u. U. einen impliziten Zusatz enthält (s. Abschn. 4.8, S. 33), der bei der Adreßberechnung in der Form

```
*relfld = *relfld + ...;
```

verwendet wird. Aus diesem Grund stellt die Relokation keine idempotente Operation dar. Um ein Objektmodul dennoch wiederholt relozieren zu können, muß der ursprüngliche Inhalt von Relokationsfeldern an anderer Stelle aufbewahrt und bei der wiederholten Durchführung von Relokationen anstelle des aktuellen Inhalts der Relokationsfelder verwendet werden.

### 6.1.6 Versionsverwaltung

Der implementierte dynamische Linker realisiert eine einfache Versionsverwaltung auf der Basis einer Major-Versionsnummer *maj* und einer Minor-Versionsnummer *min* eines Objektmoduls. Die neuere

[3] Als Beispiel sei hier auf `libc.so` verwiesen, die unter Solaris 2.x weit über 1000 Relokationsinformationen enthält.

von zwei Versionen eines Objektmoduls wird durch eine höhere Major-Versionsnummer oder eine höhere Minor-Versionsnummer bei gleicher Major-Versionsnummer gekennzeichnet. Die gleiche Major-Versionsnummer zeigt die Kompatibilität beider Versionen an. Ziel der Versionsverwaltung ist es, unter Wahrung der Kompatibilität nach Möglichkeit die neuere von zwei Versionen eines Objektmoduls zu verwenden, während die ältere aus dem Hauptspeicher entfernt oder aber nicht in den Hauptspeicher geladen wird. Maßnahmen der Versionsverwaltung kommen während der *link*- und *relink*-Operation zum Einsatz.

Für die *link*-Operation werden dazu die in Tabelle 6.1 dargestellten Aktionen implementiert. Hierbei steht $M$ für die bereits geladene Version eines Objektmoduls und $M'$ für die Version des Objektmoduls, die mit der Erstellung des Abhängigkeitsgraphen $G_i$ geladen werden soll. Die Paare $(maj, min)$ bzw. $(maj', min')$ bezeichnen die Versionen von $M$ bzw. $M'$.

| $maj \leftrightarrow maj'$ | $min \leftrightarrow min'$ | Aktion |
|---|---|---|
| $\neq$ | $*$ | $M$ wird weiterverwendet<br>$M'$ wird in $G_i$ verwendet |
| $=$ | $<$ | $M'$ ersetzt $M$ |
| | $\geq$ | $M$ wird statt $M'$ verwendet |

Tabelle 6.1: Versionsverwaltung während der *link*-Operation

Durch die in Tabelle 6.1 dargestellte Vorgehensweise wird sichergestellt, daß jeder Abhängigkeitsgraph die neueste kompatible Version eines Objektmoduls benutzt. Aus diesem Grund erübrigt sich während der *relink*-Operation eines Abhängigkeitsgraphen $G_i$ die Überprüfung, ob eine neuere kompatible Version eines Objektmoduls bereits im Hauptspeicher existiert. Objektmodule werden deshalb während einer *relink*-Operation unbedingt neu geladen. Die Versionsverwaltung stellt anschließend sicher, daß alle Abhängigkeitsgraphen $G_j$ $(i \neq j)$ Verweise auf ein Objektmodul $M$ durch Verweise auf eine neuere kompatible Version $M'$ des gleichen Objektmoduls ersetzen. Damit ändert sich die Vorgehensweise während einer *relink*-Operation auf die in Tabelle 6.2 dargestellte Weise.

| $maj \leftrightarrow maj'$ | $min \leftrightarrow min'$ | Aktion |
|---|---|---|
| $\neq$ | $*$ | $M$ wird weiterverwendet<br>$M'$ wird in $G'_i$ verwendet |
| $=$ | $\leq$ | $M'$ ersetzt $M$ |
| | $>$ | $M$ wird weiterverwendet<br>$M'$ wird in $G'_i$ verwendet |

Tabelle 6.2: Versionsverwaltung während der *relink*-Operation

### 6.1.7 Typsicherheit

Durch das ELF-Objektdateiformat werden keine Typinformationen für Symbole verwaltet. Typsicherheit kann daher nur durch die Übereinstimmung entsprechend gestalteter Symbolnamen erreicht werden. Dafür wird vorausgesetzt, daß die Eingabe-Objektmodule des dynamischen Linkers durch einen C++-Compiler erzeugt wurden. Ein solcher Compiler erzeugt dekorierte Symbolnamen, durch die die Relokation von Funktionsaufrufen typsicher gestaltet werden kann. Fordert der Nutzer des dynamischen Linkers jedoch die Adresse einer Funktion an, setzt dies voraus, daß er den Mechanismus des Dekorierens von Symbolnamen (*Mangling*) kennt. Diese Vorgehensweise ist wenig nutzerfreundlich, da das Mangling ein komplexer Mechanismus ist, der sich zudem von Compiler zu Compiler unterscheidet.

Ein erster Lösungsansatz dieses Problems besteht darin, daß der Nutzer die gesuchte Funktion in Form eines Prototypnamens $p$ der Form

```
prototypename := functionname "(" typename {"," typename} ")" .
```

angibt. Durch eine Mangling-Funktion $M$ kann $p$ in einen dekorierten Symbolnamen übersetzt und bei der anschließenden Symbolsuche mit dem Namen $n$ einer Symboldefinition verglichen werden. Durch die Komplexität des Mangling-Mechanismus[4] gestaltet sich die Konstruktion der Funktion $M$ jedoch äußerst schwierig.

Ein zweiter Lösungsansatz ergibt sich aus dem Fakt, daß ein Vergleich von $n$ und $p$ auch über eine zu $M$ inverse Demangling-Funktion $D$ möglich ist:

$$n = M(p) \Leftrightarrow p = D(n)$$

Eine solche Funktion $D$ ist auf den untersuchten Plattformen als Bestandteil verschiedener Programme (z. B. `c++filt(1)` oder `dem(1)`) verfügbar. Im Fall der vorliegenden Implementation wird hierfür die Funktion `cplus_demangle()` des zur GNU-C++-Compiler-Distribution gehörenden Quelltext-Moduls `cplus-dem.c` verwendet. Zur Effektivitätssteigerung wird jede Konvertierung $D(n)$ nur einmal ausgeführt (und zwar beim Laden eines Objektmoduls) und das Ergebnis in der *Tabelle von Prototypnamen* (Instanzvariable `dmgsnm` der Klasse `File`) abgelegt. Diese Tabelle ist indexgleich zur normalen Symboltabelle und damit kann bei der Symbolsuche der Vergleich zwischen Symbolnamen mit Hilfe der Einträge dieser Tabelle erfolgen.

### 6.1.8 Symbole der Linker-Applikation

Als *Linker-Applikation* wird das Nutzerprogramm bezeichnet, das die Dienste des dynamischen Linkers benutzt. Um dessen Symboldefinitionen verfügbar zu machen, sind auf der zugrundeliegenden Plattform die folgenden beiden Vorgehensweisen denkbar:

1. Die Symboltabelle für das dynamische Linken (`.dynsym`) der Linker-Applikation wird benutzt.
   Die Linker-Applikation selbst wird als dynamisch gelinktes Programm implementiert. In diesem Fall enthält die Linker-Applikation eine Symboltabelle für das dynamische Linken (`.dynsym`-Sektion), die vom Programmlader mit in den Hauptspeicher eingebracht wird. Der Programmlader legt im Stack der Linker-Applikation den sogenannten *auxiliary-Vektor*[5] ab. Über diesen Vektor wird der Linker-Applikation der Zugang zur eigenen Program Header Table und damit zum eigenen Dynamic-Segment ermöglicht. Das Dynamic-Segment wiederum gibt die Position der Symboltabelle für das dynamische Linken an. Die darin enthaltenen Adressen beziehen sich auf die Basisadresse 0, die mit der tatsächlichen Basisadresse eines dynamisch gelinkten Programms übereinstimmt.

2. Die Linker-Applikation lädt ihre eigene Symboltabelle (`.symtab`-Sektion) aus der ausführbaren Datei nach.
   Eine Identifikation der ausführbaren Datei ist z. B. über den Parameter `argv[0]` der `main()`-Funktion möglich. Für die in der Symboltabelle enthaltenen Adressen gilt das gleiche wie oben.

Im Fall eines dynamisch gelinktes Programm enthält die Symboltabelle für das dynamische Linken keine Einträge für Symbole, die durch die Applikation selbst definiert werden[6]. Desweiteren gilt es zu beachten, daß bei einer Integration des dynamischen Linkers in den Kernel eines Betriebssystems kein zum *auxiliary-Vektor* äquivalenter Mechanismus existiert. Daher ist Vorgehensweise 1. zur Lösung des Problems der Kernel-Symbole unbrauchbar.

Die im Fall von Vorgehensweise 2. nachgeladene Symboltabelle ist bezüglich der Symbole, die durch die Applikation definiert werden, vollständig. Aus diesem Grund wurde in der vorliegenden Version des dynamischen Linkers diese Vorgehensweise implementiert.

[4] Die Beschreibung der Mangling-Grammatik des Sun-C++-Compilers füllt 12 Seiten, der Quelltext einer Demangling-Funktion für den GNU-C++-Compiler nimmt über 40 Seiten ein.

[5] s. `/usr/include/sys/auxv.h` bzw. `/usr/include/linux/elf.h`

[6] Diese Aussage ist eine Vereinfachung, eine vollständige Erläuterung würde über den Rahmen der hier zu treffenden Feststellungen hinausgehen.

## 6.2 Nutzerschnittstelle

Die Nutzerschnittstelle des dynamischen Linkers wird durch die Klasse `Module`, ihren öffentlichen Methoden sowie Konstanten zur Belegung der Parameter dieser Methoden gebildet. Ein Objekt dieser Klasse repräsentiert jeweils einen Abhängigkeitsgraphen. Die Funktionen `ksyminit()` und `ksymdone()` zählen ebenfalls zur Nutzerschnittstelle. Nachfolgende Auflistung beschreibt die Methoden und Funktionen der Nutzerschnittstelle, zur Erläuterung der Konstanten sei auf Anh. A.2 (S. 92) verwiesen.

- `Module::Module(char *name, unsigned int mode = MOD_GLOBAL)`

  Der Konstruktor implementiert die *link*-Operation, indem ein Abhängigkeitsgraph im Hauptspeicher beginnend bei der durch den Parameter `name` bezeichneten geteilt benutzbaren Objektdatei aufgebaut wird.

  Stellt `name` einen Pfadnamen dar, so wird versucht, die Datei unter genau diesem Namen zu öffnen. Ansonsten wird eine Objektdatei in den Verzeichnissen gesucht, die durch den Runpath des im Abhängigkeitsgraphen übergeordneten Objektmoduls bezeichnet werden. Liegt kein Runpath vor (z. B. im Fall des Objektmoduls, das die Wurzel des Abhängigkeitsgraphen darstellt) oder wird die Datei in den darin angegebenen Verzeichnissen nicht gefunden, werden die durch das Makro `STDPATH` festgelegten Verzeichnisse durchsucht. Tritt während des Ladens ein Fehler auf, so werden alle bis dahin durch das `Module`-Objekt geladenen Objektmodule gelöscht. Das `Module`-Objekt selbst bleibt bestehen, so daß durch dessen Methode `geterrno()` (s. u.) die Fehlerursache ermittelt werden kann. Vor dem Laden wird jeweils überprüft, ob das angeforderte Objektmodul bereits im Hauptspeicher enthalten ist. Wenn das der Fall ist, wird lediglich ein Verweis auf das Objektmodul erzeugt und sein Referenzzähler erhöht. Danach werden alle durch den Aufbau des Abhängigkeitsgraphen geladenen Objektmodule reloziert und ihr Initialisierungscode wird in Reihenfolge der Postorder-Breite-zuerst-Durchmusterung (Breadth First Traversal, BFT) ausgeführt (s. Abschn. 6.5.1, S. 66). Objektmodule, die durch den betrachteten Abhängigkeitsgraphen referenziert, aber nicht geladen werden, werden nicht reloziert bzw. initialisiert, da dies bereits zuvor während der Erzeugung eines anderen Abhängigkeitsgraphen geschehen ist. Durch den optionalen Parameter `mode` kann angezeigt werden, ob der erzeugte Abhängigkeitsgraph zur globalen Symbolsuche freigegeben ist. Tabelle A.2 in Anh. A.2 stellt die möglichen Werte für diesen Parameter dar.

- `Module::~Module(void)`

  Der Destruktor implementiert die *unlink*-Operation, indem – symmetrisch zum Konstruktor – ein Abhängigkeitsgraph aus dem Hauptspeicher entfernt wird. Zu diesem Zweck werden die Referenzzähler der im Abhängigkeitsgraphen enthaltenen Objektmodule dekrementiert. Der Terminierungscode von Objektmodulen, deren Referenzzähler den Wert 0 angenommen haben, wird in Reihenfolge der Preorder-BFT des Abhängigkeitsgraphen ausgeführt (s. Abschn. 6.5.1, S. 66). Anschließend werden diese Objektmodule gelöscht.

- `unsigned int Module::getsym(char *name, int mode = SNM_PROTO)`

  Diese Methode realisiert die *getsym*-Operation, indem die Adresse eines im Gültigkeitsbereich

  1. der Linker-Applikation,
  2. des zugehörigen Abhängigkeitsgraphen oder
  3. anderer, für die globale Symbolsuche freigegebener Abhängigkeitsgraphen

  definierten Symbols zurückgegeben wird. Die Gültigkeitsbereiche werden dabei in der angegebenen Reihenfolge durchsucht, wobei die Suche mit dem ersten Auffinden einer entsprechenden Symboldefinition abgebrochen wird. Kann das Symbol nicht gefunden werden, so wird der Wert 0 zurückgeliefert.

  Durch den Parameter `name` wird dabei der Name des gesuchten Symbols angegeben. Durch den optionalen Parameter `mode` kann der Methode angezeigt werden, ob der Symbolname in Form eines Funktionsprototyps oder eines dekorierten Symbolnamens angegeben wurde (s. Tabelle A.3 in Anh. A.2). Die Namen von Funktionsprototypen bilden sich dabei nach dem in

Abschn. 6.1.7 (S. 59) angegebenen Schema. Die Bildung dekorierter Symbolnamen hängt vom Mangling-Mechanismus des verwendeten Compilers ab[7]. Datenobjekte werden unabhängig von der Art der Namensbildung gefunden.

Dem Nutzer obliegt es, den Rückgabewert der Methode in einer für ihn geeigneten Weise (z. B. durch einen entsprechenden cast) zu interpretieren.

- `int Module::geterrno(void)`

  Tritt während des Ladens oder der Relokation von Objektmodulen ein Fehler auf, so kann mit dieser Methode eine Fehlernummer abgefragt werden. In Tabelle A.4 in Anh. A.2 sind die möglichen Fehlernummern und ihre Bedeutung dargestellt.

- `char *Module::geterrmsg(void)`

  Wenn ein Fehler aufgetreten ist, so liefert diese Methode einen Zeiger auf einen kurzen Text zur Fehlerbeschreibung zurück. Ist kein Fehler aufgetreten, so wird der Wert NULL zurückgegeben.

- `char *Module::getname(void)`

  Zur Identifikation eines Abhängigkeitsgraphen wird durch diese Methode ein Zeiger auf den Namen des Objektmoduls, das die Wurzel des Graphen markiert, zurückgegeben.

- `void Module::dispinfo(int mode = DSP_FILES | DSP_SCOPE)`

  Durch diese Methode werden Informationen über alle Abhängigkeitsgraphen bzw. alle Objektmodule, die im Hauptspeicher enthalten sind, auf die Standardausgabe geschrieben. Durch den Parameter `mode` kann festgelegt werden, welche Art von Information ausgegeben werden soll. Tabelle A.5 in Anh. A.2 listet die möglichen Werte für `mode` und deren Bedeutung auf.

- `void Module::relink(void)`

  Diese Methode realisiert die *relink*-Operation. Hiermit kann bewirkt werden, daß die zum Abhängigkeitsgraphen gehörenden Objektmodule erneut aus den entsprechenden Dateien in den Hauptspeicher geladen werden. Auf diese Weise wird eine neue Version eines Abhängigkeitsgraphen im Hauptspeicher erstellt. Nachdem dies geschehen ist, wird der Terminierungscode der Objektmodule des alten Abhängigkeitsgraphen ausgeführt und anschließend werden diese Objektmodule gelöscht. Danach findet die Relokation und die Ausführung des Initialisierungscodes der Objektmodule des neuen Abhängigkeitsgraphen statt. Als letzter Schritt erfolgt eine Re-Relokation aller Objektmodule, die Symboldefinitionen aus gelöschten Objektmodulen benutzt haben.

- `void ksyminit(char *name)`

  Durch diese Funktion wird die Symboltabelle der Linker-Applikation (s. Abschn. 6.1.8, S. 60) verfügbar gemacht. Zu diesem Zweck muß mit dem Parameter `name` der Name der ausführbaren Datei der Linker-Applikation angegeben werden. Hierzu kann der `argv[0]`-Parameter der `main()`-Funktion der Linker-Applikation verwendet werden. Der Aufruf von `ksyminit()` muß noch vor der Instantiierung eines Objekts der Klasse `Module` erfolgen.

- `void ksymdone(void)`

  Wird der dynamische Linker nicht mehr benötigt, kann mittels dieser Funktion der für die Symboltabelle der Linker-Applikation benötigte Speicherplatz wieder freigegeben werden.

Um die Nutzerschnittstelle des dynamischen Linkers in einem Quelltextmodul bekannt zu machen, muß dieses Quelltextmodul die Datei `odl.H` einschließen. Eine Linker-Applikation wird anschließend mit der Compiler-Option `-lodl` erzeugt. In Anh. A.5 (S. 97) ist ein Beispiel für die Verwendung der Nutzerschnittstelle des dynamischen Linkers angegeben.

[7] In der vorliegenden Version des dynamischen Linkers wird das Mangling des GNU-C++-Compilers (gcc 2.7.2) unterstützt.

## 6.3 Systemschnittstelle

Beim Entwurf des dynamischen Linkers galt es, eine spätere Migration des Programms in eine Stand-Alone-Umgebung zu berücksichtigen. Aus diesem Grund ist besonderes Augenmerk darauf zu richten, welche Dienste der Betriebssystemumgebung für die Arbeit des Linkers genutzt werden. Tabelle A.7 (s. Anh. A.3, S. 94 listet zu diesem Zweck alle Bibliotheksfunktionen auf, die durch den dynamischen Linker verwendet werden. In der gegenwärtigen Implementation wird der Name der ausführbaren Datei der Linker-Applikation benötigt. Hierfür wird der `argv[0]`-Parameter der `main()`-Funktion der Linker-Applikation benutzt. Der dynamische Linker verwendet darüber hinaus keine Umgebungsvariablen.

## 6.4 Datenstrukturen

Der folgende Abschn. beschreibt die dem dynamischen Linker zugrundeliegenden Datenstrukturen. Hierbei wird vor allem Wert auf globale Zusammenhänge gelegt, Implementationsdetails sollten im Quelltext nachgelesen werden.

### 6.4.1 Klasse Module

Die Klasse `Module` formt die Nutzerschnittstelle des dynamischen Linkers. Jedes Objekt dieser Klasse repräsentiert einen Abhängigkeitsgraphen. Durch den Konstruktor und Destruktor dieser Klasse wird die *link*- bzw. die *unlink*-Operation realisiert. Weitere Methoden dieser Klasse implementieren die *getsym*- und *relink*-Operation.

Alle Objekte der Klasse `Module` sind über die Instanzvariable `next` in einer Liste, der sogenannten *Module Chain*, verkettet. Die Reihenfolge ihrer Einträge entspricht der umgekehrten Reihenfolge der Erzeugung der `Module`-Objekte. Der Beginn dieser Liste wird dabei durch die Klassenvariable `Module::chain` gekennzeichnet (s. Abb. 6.7).

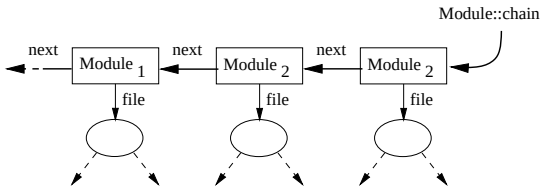


Abbildung 6.7: Module Chain

Durch die Verkettung aller `Module`-Objekte werden Aktionen ermöglicht, die über den einzelnen Gültigkeitsbereich hinausgehen. Das sind z. B.

- die Aktualisierung von Gültigkeitsbereichen und die Re-Relokation von Objektmodulen im Fall einer *link*- oder *relink*-Operation,
- die Anzeige von Statusinformationen des dynamischen Linkers und
- die globale Symbolsuche.

Jedes Objekt der Klasse `Module` verweist über die Instanzvariable `file` auf ein Objekt der Klasse `File`. Dieses `File`-Objekt repräsentiert das Objektmodul, welches die Wurzel des Abhängigkeitsgraphen darstellt.

### 6.4.2 Klasse File

Die Knoten von Abhängigkeitsgraphen werden durch Objekte der Klasse `File` gebildet. Ein Objekt der Klasse `File` repräsentiert dabei genau ein Objektmodul. Umgekehrt existiert für ein Objektmodul genau ein `File`-Objekt. Im vorliegenden Fall ist unter einem Objektmodul die Hauptspeicherrepräsentation des Inhalts einer geteilt benutzbaren Objektdatei zu verstehen. Ein `File`-Objekt beschreibt ein solches Objektmodul durch seine Basisadresse (Instanzvariable `base`) und seine Größe (Instanzvariable `size`). Darüber hinaus werden die folgenden Abschnitte eines Objektmoduls bezeichnet:

- Dynamic-Segment
- Symboltabelle
- Relokationstabellen
- String-Tabelle
- Hash-Tabellen

Die dafür verwendeten Instanzvariablen sind in Abb. 6.8 dargestellt.

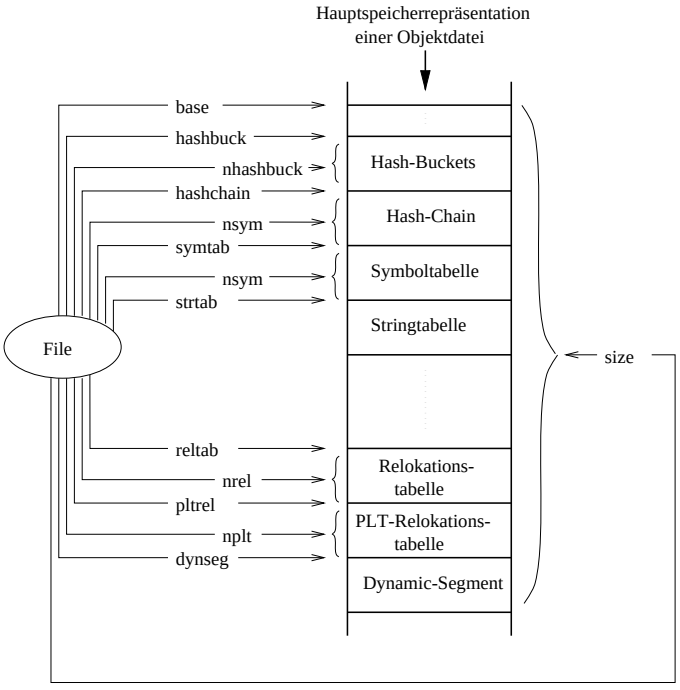


Abbildung 6.8: Instanzvariablen der Klasse `File`

Alle Objekte der Klasse `File` sind – analog zu `Module`-Objekten – über die Instanzvariable `next` in einer Liste, der sog. *File Chain*, verkettet. Die Reihenfolge ihrer Einträge entspricht der umgekehrten Reihenfolge, in der die Objektmodule geladen wurden. Der Beginn dieser Liste wird dabei durch die Klassenvariable `File::chain` gekennzeichnet (s. Abb. 6.9). Durch die Verkettung aller `File`-Objekte kann auf effektive Weise überprüft werden, ob ein Objektmodul bereits im Hauptspeicher existiert. Die Instanzvariable `nref` stellt den Referenzzähler eines Objektmoduls dar.

Ein `File`-Objekt verweist durch seine *Abhängigkeitsliste* (Instanzvariable `deplst`) auf seine Nachfolger im Abhängigkeitsgraphen. Anzahl und Reihenfolge der Einträge der Abhängigkeitsliste entspricht dabei direkt den `DT_NEEDED`-Einträgen im Dynamic-Segment des Objektmoduls. Die Liste effektiv nutzender und effektiv genutzter Objektmodule (s. Abschn. 6.1.5, S. 58) wird mittels der Instanzvariablen `usrlst` und `reflst` realisiert.

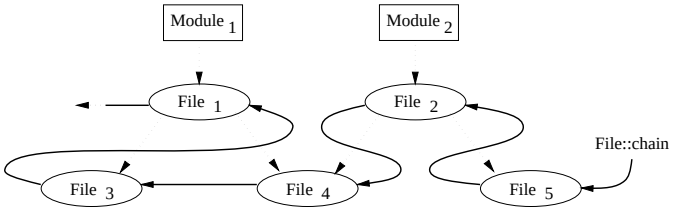


Abbildung 6.9: File Chain

### 6.4.3 Klasse Filelist

Objekte der Klasse `Filelist` werden verwendet, um von einem `File`-Objekt ausgehend beliebig viele Verweise auf andere `File`-Objekte verwalten zu können. Objekte der Klasse `Filelist` bilden über die Instanzvariable `next` eine vorwärtsverkettend aufgebaute Liste und verweisen über die Instanzvariable `file` auf das zu referenzierende `File`-Objekt (s. Abb. 6.10).

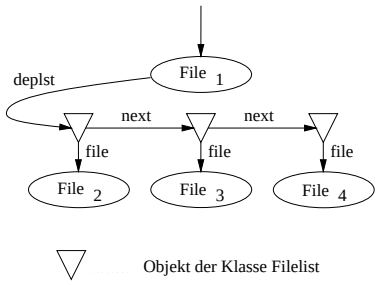


Abbildung 6.10: Verwendung von `Filelist`-Objekten

Sowohl Abhängigkeitslisten als auch die Listen effektiv nutzender und effektiv genutzter Objektmodule werden durch Instanzen der Klasse `Filelist` realisiert. Eine Dopplung gleicher Einträge ist in keiner dieser Listen sinnvoll und wird daher durch die Methoden der Klasse `Filelist` verhindert.

### 6.4.4 Symboltabelle der Linker-Applikation

Die Symboltabelle der Linker-Applikation ist eine Kopie der `.symtab`-Sektion der ausführbaren Datei der Linker-Applikation. Ihre Adresse und die Anzahl ihrer Einträge wird durch die im Quelltextmodul `ksym.C` auf Dateiebene definierten Variablen `ksymtab` und `knsym` beschrieben. Da Symbole durch ihre Namen identifiziert werden und Symbolnamen in einer Stringtabelle enthalten sind, wird neben der Symboltabelle auch die Stringtabelle aus der ausführbaren Datei der Linker-Applikation in den Hauptspeicher geladen und ihre Adresse durch die Variable `kstrtab` markiert.

## 6.5 Algorithmen

Um ein grundsätzliches Verständnis der Funktionsweise des dynamischen Linkers zu ermöglichen, wird im folgenden Abschn. eine Auswahl zugrundeliegender Algorithmen beschrieben. Als Ausdrucksmittel wird dabei – wo angebracht – eine Pseudocode-Notation eingesetzt. Im Rahmen dieser Notation werden die folgenden Kürzel verwendet:

| | | |
|---|---|---|
| $e$ | ... | Wurzel eines Abhängigkeitsgraphen |
| $k$ | ... | der während der Durchmusterung eines Abhängigkeitsgraphen betrachtete Knoten |
| $n$ | ... | Name eines Objektmoduls oder einer Objektdatei |
| $G$ | ... | Identifikation für einen Abhängigkeitsgraphen |
| $S_G$ | ... | Menge der Symboldefinitionen eines Abhängigkeitsgraphen $G$ |
| $L, K$ | ... | Listen von Knoten |
| $\{\}$ | ... | leere Knotenliste |
| $A$ | ... | Aktion, die bei der Durchmusterung eines Abhängigkeitsgraphen über jedem Knoten auszuführen ist |

### 6.5.1 Breite-zuerst-Durchmusterung eines Abhängigkeitsgraphen

Die Breite-zuerst-Durchmusterung (Breadth First Traversal, BFT) wird immer dann eingesetzt, wenn die entsprechende Besuchsreihenfolge von Bedeutung ist, so z. B. bei

- der Symbolsuche,
- der Ausführung von Initialisierungs- und Terminierungscode,
- der Anzeige von Statusinformationen oder
- dem Löschen von Objektmodulen.

Zu diesem Zweck wird der in [Dilger93] vorgestellte BFT-Algorithmus verwendet. Ein Problem ergibt sich hierbei jedoch für mehrfach referenzierte Knoten, die durch diesen Durchmusterungsalgorithmus wiederholt besucht würden. Zur Lösung dieses Problems wird der Durchmusterungsalgorithmus um die Verwaltung einer Liste bereits betrachteter Knoten erweitert. Ein Knoten wird nur dann besucht, wenn er nicht in dieser Liste ($K$) enthalten ist. Algorithmus 6.1 stellt die sich daraus ergebende Vorgehensweise dar. Der Start des Algorithmus erfolgt durch $B(e, A, \{\}, \{\})$.

$B(k, A, K, L)$:

1. führe $A$ über $k$ aus
2. hänge $k$ an $K$ an
3. für jeden Nachfolger $n$ von $k$
   wenn $n$ nicht in $K$ enthalten ist
   hänge $n$ an $L$ an
4. wenn $L$ nicht leer ist
   extrahiere den ersten Knoten $l$ aus $L$
   rufe $B(l, A, K, L)$
5. kehre zurück

**Algorithmus 6.1:** Breite-zuerst-Durchmusterung eines Abhängigkeitsgraphen

Durch Algorithmus 6.1 wird die Aktion $A$ in Preorder-Reihenfolge über den Knoten eines Abhängigkeitsgraphen ausgeführt (Preorder-BFT). Durch Verschiebung des Schritts 1. hinter den Schritt 4. kann eine Postorder-Reihenfolge der Ausführung von $A$ erreicht werden (Postorder-BFT).

Ein Objektmodul kann durch die BFT mehrerer Abhängigkeitsgraphen wiederholt besucht werden. Dabei gilt es zu beachten, daß Aktionen wie der Aufruf des Initialisierungscodes eines Objektmoduls nur einmal ausgeführt werden dürfen. Für eine solche Aktion wird ein Flag verwaltet. Sie wird nur dann ausgeführt, wenn das entsprechende Flag nicht gesetzt ist. Im Anschluß an die Ausführung der Aktion wird das Flag gesetzt.

### 6.5.2 Tiefe-zuerst-Durchmusterung eines Abhängigkeitsgraphen

Die Tiefe-zuerst-Durchmusterung (Depth First Traversal, DFT) wird in den Fällen verwendet, wo die Besuchsreihenfolge der Knoten eines Abhängigkeitsgraphen keine Rolle spielt, so z. B. bei

- der Relokation von Objektmodulen,
- der Dekrementierung von Referenzzählern oder
- der Aktualisierung von Referenzen nach einer *link*- oder *relink*-Operation.

Zu diesem Zweck wird die durch Algorithmus 6.2 skizzierte Vorgehensweise verwendet. Der Start des Algorithmus erfolgt durch $T(e, A)$.

$T(k, A)$:

1. führe $A$ über $k$ aus
2. für jeden Nachfolger $n$ von $k$
   rufe $T(n, A)$
3. kehre zurück

**Algorithmus 6.2:** Tiefe-zuerst-Durchmusterung eines Abhängigkeitsgraphen

Auch hier bewirkt die Tatsache, daß der Abhängigkeitsgraph ein gerichteter, azyklischer Graph und kein Baum ist, daß Knoten durch diesen Algorithmus mehrfach besucht werden. Dies ist jedoch – im Gegensatz zur BFT – ein in einigen Fällen nützlicher Umstand. Als Beispiel sei hier die Dekrementierung von Referenzzählern im Fall der Beseitigung eines Abhängigkeitsgraphen genannt. So wird z. B. der Referenzzähler von $D$ aus Abb. 6.1 (s. S. 54) in zwei Schritten – einmal von $A$ und einmal von $B$ aus – zu 0 dekrementiert. Soll eine nur einmal auszuführende Aktion (z. B. die Relokation eines Objektmoduls) mit Hilfe der DFT über den Knoten eines Graphen durchgeführt werden, kommt der bereits bei der BFT erwähnte Flag-Mechanismus zum Einsatz.

### 6.5.3 Erzeugung eines Abhängigkeitsgraphen

Ein Abhängigkeitsgraph wird im Speicher nach der durch Algorithmus 6.3 skizzierten Vorgehensweise in Tiefe-zuerst-Strategie aufgebaut. Der Start des Algorithmus erfolgt mit $create(n_e)$, wobei $n_e$ der Name des Objektmoduls ist, das die Wurzel des Abhängigkeitsgraphen darstellt.

*create*$(n)$:

wenn ein Knoten $k$ mit dem Namen $n$ existiert

1. inkrementiere Referenzzähler von $k$
2. gib $k$ zurück

sonst

1. erzeuge $k$ durch das Laden einer Datei namens $n$
2. initialisiere Referenzzähler von $k$ mit 1
3. initialisiere Abhängigkeitsliste von $k$ mit {}
4. für jede Abhängigkeit $a$ von $k$
   füge Rückgabewert von $create(n_a)$ an die Abhängigkeitsliste von $k$ an
5. gib $k$ zurück

**Algorithmus 6.3:** Erzeugung eines Abhängigkeitsgraphen

Der Algorithmus wird durch den Konstruktor der Klasse `Module` angestoßen. Die Methode `init()` der Klasse `File` realisiert dabei die Rekursion, während die Methode `load()` Verwendung findet, um eine geteilt benutzbare Objektdatei segmentweise in den Hauptspeicher zu laden. Zur Überprüfung, ob ein Objektmodul im Hauptspeicher enthalten ist, wird durch die Methode `checkchain()` die *File Chain* (s. Abschn. 6.4.2, S. 64) durchmustert. Nach dem Aufbau des Abhängigkeitsgraphen werden die dadurch geladenen Objektmodule reloziert (s. Abschn. 6.5.5, S. 68). Anschließend wird durch die Methode `initialize()` der Klasse `File` mittels Postorder-BFT der Initialisierungscode dieser Objektmodule zur Ausführung gebracht.

### 6.5.4 Löschen eines Abhängigkeitsgraphen

Beim Löschen eines Abhängigkeitsgraphen gilt es zu beachten, daß nur die darin enthaltenen Objektmodule, die nicht durch andere Abhängigkeitsgraphen referenziert werden, zu löschen sind. Aus diesem Grund werden zuerst die Referenzzähler der im betrachteten Abhängigkeitsgraphen enthaltenen Objektmodule dekrementiert. Ein Objektmodul kann anschließend gelöscht werden, wenn sein Referenzzähler den Wert 0 angenommen hat. Die Dekrementierung geschieht zweckmäßigerweise mittels DFT, da hierbei Referenzzähler mehrfach referenzierter Objektmodule (z. B. $D$ in Abb. 6.1, S. 54) wiederholt dekrementiert werden.

Eine Besonderheit bilden dabei die Nachfolger mehrfach referenzierter Knoten. Die Referenzzähler dieser Module würden bei der in Abschn. 6.5.2 (S. 67) dargestellten DFT voreilig zu 0 oder aber zu einem negativen Wert dekrementiert. Als Beispiel sei hier auf Objektmodul $F$ aus Abb. 6.2 (s. S. 55) verwiesen, dessen Referenzzähler beim Löschen des Abhängigkeitsgraphen $G_1$ zu 0 dekrementiert würde, obwohl es als Bestandteil des Abhängigkeitsgraphen $G_2$ erhalten bleiben muß.

Aus diesem Grund wird die durch Algorithmus 6.4 skizzierte Vorgehensweise implementiert. Der Start des Algorithmus erfolgt durch $decrefs(e)$.

$decrefs(k)$:

1. dekrementiere Referenzzähler von $k$
2. wenn Referenzzähler von $k = 0$
   für jeden Nachfolger $n$ von $k$
   rufe $decrefs(n)$
3. kehre zurück

**Algorithmus 6.4:** Dekrementierung von Referenzzählern

Durch den Destruktor der Klasse `Module` wird mit der Methode `decrefs()` der Klasse `File` der oben beschriebene Algorithmus angestoßen. Danach wird durch die Methode `terminate()` der Klasse `File` in einer Preorder-BFT des Graphen der Terminierungscode all jener Objektmodule ausgeführt, deren Referenzzähler den Wert 0 besitzt. Anschließend werden diese Objektmodule durch die Methode `done()` mittels einer Postorder-BFT gelöscht.

### 6.5.5 Relokation

Ein Objektmodul wird stets innerhalb des Abhängigkeitsgraphen reloziert, durch den es geladen wurde. Da die Reihenfolge der Relokation von Objektmodulen innerhalb eines Abhängigkeitsgraphen keine Rolle spielt, kann hierfür eine DFT verwendet werden. Vom dynamischen Linker wird dazu die Methode `relocate()` der Klasse `File` benutzt. Eine wiederholte Relokation von mehrfach referenzierten Objektmodulen oder Objektmodulen, die durch einen anderen Abhängigkeitsgraphen geladen wurden, wird dabei in der oben dargestellten Weise mittels eines speziellen Flags verhindert.

Durch das Dynamic-Segment einer geteilt benutzbaren Objektdatei werden die Adressen zweier Relokationstabellen, einer für PLT-Relokationen und einer für alle übrigen Relokationen, angegeben.

Da der dynamische Linker in der vorliegenden Implementation kein verzögertes Binden realisiert, werden beide Arten von Relokationen durch `relocate()` unmittelbar hintereinander ausgeführt. Die in den Relokationsinformationen angegebenen Adressen der Relokationsfelder beziehen sich auf die Basisadresse 0. Somit kann die Position der zu modifizierenden Speicherzelle durch Addition der tatsächlichen Basisadresse eines Objektmoduls berechnet werden. Zur Wahrung der Idempotenz von Relokationen (s. Abschn. 6.1.5, S. 58) wird unmittelbar nach dem Laden eines Objektmoduls Speicher für die Aufbewahrung des ursprünglichen Inhalts von Relokationsfeldern allokiert. Dieser Speicherbereich wird als Feld organisiert und durch die Nummer der entsprechenden Relokationsinformation indiziert. Die Instanzvariable `orgrel` der Klasse `File` verweist auf dieses Feld.

Zur Durchführung symbolischer Relokationen muß innerhalb des Abhängigkeitsgraphen nach der entsprechenden Symboldefinition gesucht werden. Aus diesem Grund wird von `relocate()` der Zeiger auf das den Abhängigkeitsgraphen identifizierende `Module`-Objekt mitgeführt, dessen private Methode `getsym()` zur Symbolsuche benutzt wird. Abbildung 6.11 stellt an einem Beispiel den Vorgang der Durchmusterung eines Abhängigkeitsgraphen zur Durchführung einer symbolischen Relokation dar. Hierbei wird durch die DFT das zu relozierende Objektmodul $File_4$ erreicht (Schritte 1, 2 und 3). Bei der Ausführung einer symbolischen Relokation von $File_4$ wird die private `getsym()`-Methode von $Module_1$ aufgerufen (Schritt 4), die den Abhängigkeitsgraphen mittels Breitensuche durchmustert. Dabei werden die Objektmodule $File_1$ und $File_2$ durchsucht, bevor die benötigte Symboldefinition in $File_3$ gefunden wird (Schritte 5, 6 und 7).

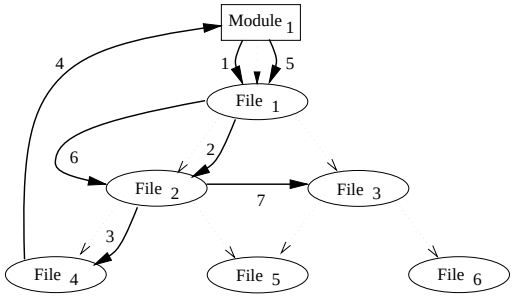


Abbildung 6.11: Relokation und Symbolsuche

Im Zug der Symbolsuche wird neben der Adresse des Symbolwerts ein Zeiger auf das Objektmodul zurückgegeben, welches das Symbol definiert. Über diesen Zeiger wird die wechselseitige Registrierung von nutzendem und bereitstellendem Objektmodul mittels der Listen effektiv nutzender und effektiv genutzter Objektmodule vorgenommen (s. Abschn. 6.1.5, S. 58).

### 6.5.6 Symbolsuche

Die Suche nach Symboldefinitionen wird zur Durchführung symbolischer Relokationen und auf Nutzeranforderung hin eingeleitet. Im ersten Fall wird eine benötigte Funktion durch einen dekorierten Symbolnamen bezeichnet. Mit diesem Symbolnamen kann die Symboltabelle eines Objektmoduls unter Zuhilfenahme eines Hashing-Algorithmus durchsucht werden. Für Datenobjekte gilt das gleiche, im Gegensatz zu Funktionen verfügen sie jedoch nicht über einen dekorierten Symbolnamen. Im zweiten Fall besteht für den Nutzer des dynamischen Linkers die Möglichkeit, eine gesuchte Funktion durch einen Prototypnamen (s. Abschn. 6.1.7, S. 59) zu beschreiben. Hierbei wird innerhalb eines Objektmoduls die Tabelle von Prototypnamen in Form der Instanzvariable `dmgsnm` der Klasse `File` durchsucht, und durch deren Index auf die Position innerhalb der Symboltabelle geschlossen. Datenobjekte können sowohl über die Tabelle von Prototypnamen als auch direkt über die Symboltabelle lokalisiert werden. Startpunkt einer Symbolsuche ist stets das den Abhängigkeitsgraphen identifizierende `Module`-Objekt. Durch dessen private Methode `getsym()` wird die durch Algorithmus 6.5 skizzierte Vorgehensweise realisiert. Damit in einem Objektmodul Funktionen der Linker-Applikation in Anspruch genommen werden können (z. B. die des dynamischen Linkers selbst), muß die Symboltabelle der Linker-Applikation durchsucht werden. Kann eine Symboldefinition nicht innerhalb eines

Abhängigkeitsgraphen gefunden werden, so wird die globale Symbolsuche eingeleitet, bei der alle entsprechend markierten Abhängigkeitsgraphen durchmustert werden.

*getsym*$(s, G_i)$:

kann $s$ in der Symboltabelle der Linker-Applikation gefunden werden
gib die Adresse des Symbolwerts von $s$ zurück
sonst
kann $s$ in $S_{G_i}$ gefunden werden
gib die Adresse des Symbolwerts von $s$ zurück
sonst
für jeden zur globalen Symbolsuche freigegebenen Abhängigkeitsgraphen $G_j \neq G_i$
kann $s$ in $S_{G_j}$ gefunden werden
gib die Adresse des Symbolwerts von $s$ zurück
sonst
gib die Adresse NULL zurück

**Algorithmus 6.5:** Symbolsuche

Die Suche innerhalb der Symbolmenge $S_G$ eines Abhängigkeitsgraphen $G$ wird durch die Methode `getsym()` des `File`-Objekts eingeleitet, das die Wurzel des Abhängigkeitsgraphen darstellt. Durch diese Methode wird eine Breitensuche im Graphen realisiert, wobei die Durchmusterung mit dem Auffinden einer Symboldefinition abgebrochen wird. Eine Ausnahme bildet hierbei das Auffinden einer schwach gebundenen Symboldefinition: die Adresse $a_W$ des Symbolwerts wird vermerkt und der Abhängigkeitsgraph wird weiter durchmustert. Nur wenn keine global gebundene Symboldefinition gefunden wurde, wird $a_W$ zurückgegeben.

### 6.5.7 Relinken

Eine *relink*-Operation wird durch den Aufruf der Methode `relink()` der Klasse `Module` angestoßen. Hierbei wird der zu einem `Module`-Objekt gehörende Abhängigkeitsgraph $G_i$ neu aufgebaut. Dies erfolgt gemäß der in Abschn. 6.5.3 (S. 67) beschriebenen Weise. Der einzige Unterschied besteht darin, daß im Fall einer *relink*-Operation der anfängliche Test aus Algorithmus 6.3 nicht erfolgt, und damit Objektdateien unbedingt neu geladen werden.

Nach dem Aufbau des neuen Abhängigkeitsgraphen $G_i{}'$ werden alle anderen Abhängigkeitsgraphen $G_j$ $(i \neq j)$ durchmustert und deren Referenzen zu Objektmodulen aus $G_i$ in Referenzen auf die entsprechenden Objektmodule aus $G_i{}'$ umgewandelt. Dieser Aktualisierungsprozeß wird durch die Methoden `update()` der Klassen `Module` und `File` realisiert. Die Methode der letztgenannten Klasse implementiert hierfür die durch Algorithmus 6.6 skizzierte Vorgehensweise. Der Start des Algorithmus erfolgt durch *update*$(e, G_i{}')$.

Nach der Aktualisierung aller Referenzen werden die Objektmodule des alten Abhängigkeitsgraphen $G_i$ in der in Abschn. 6.5.4 (S. 68) beschriebenen Weise gelöscht. Dabei werden die durch die Listen effektiv nutzender Objektmodule bezeichneten Knoten für eine Re-Relokation markiert. Im Anschluß daran wird der neue Abhängigkeitsgraph $G_i{}'$ reloziert und der Prozeß der Re-Relokation aller übrigen Graphen angestoßen (s. Abschn. 6.1.5, S. 58). Zum Abschluß wird der in den Objektmodulen von $G_i{}'$ enthaltene Initialisierungscode ausgeführt.

## 6.6 Probleme und Verbesserungsvorschläge

Die folgende Aufzählung stellt verschiedene bekannte Probleme beim Einsatz des dynamischen Linkers zusammen, die mit den implementierten Datenstrukturen und Algorithmen nicht oder nur unverhält-

*update*$(k, G_i')$:

wenn $G_i'$ eine neue Version $k'$ von $k$ enthält

1. lösche den durch $k$ bezeichneten Subgraphen
2. ersetze die Referenz auf $k$ durch eine Referenz auf $k'$
3. inkrementiere den Referenzzähler von $k'$

sonst

für jeden Nachfolger $n$ von $k$
rufe *update*$(n, G_i')$

**Algorithmus 6.6:** Aktualisierung von Abhängigkeitsgraphen nach einer *relink*-Operation

nismäßig schwer zu lösen sind und daher vom Nutzer zu berücksichtigen sind.

- **Benutzung von Symboldefinitionen aus Objektmodulen, die nicht unmittelbar oder mittelbar referenziert werden**
  Ein Objektmodul $M$ wird innerhalb des Abhängigkeitsgraphen $G_i$, durch den es geladen wurde, reloziert. Hierbei können Symboldefinitionen aus Objektmodulen verwendet werden, die von $M$ weder unmittelbar (Nachfolger) noch mittelbar (Nachfolger der Nachfolger) referenziert werden. Später kann $M$ zum Bestandteil eines weiteren Abhängigkeitsgraphen $G_j$ werden. Ein Beispiel hierfür ist Modul $D$ aus Abb. 6.2 (S. 55), das innerhalb des Abhängigkeitsgraphen $G_1$ reloziert wird, aber auch zum Abhängigkeitsgraphen $G_2$ gehört. Folgende Konstellationen sollen zur Darstellung der auftretenden Probleme dienen:

  1. Objektmodul $D$ definiere eine generische Sortierfunktion `sort()`, und erwartet hierfür, daß durch das nutzende Objektmodul die Ordnungsrelation in Form einer Funktion `rel()` definiert werde. Objektmodul $A$ definiert `rel()` mittels der $<$-Relation, $B$ mittels der $>$-Relation. Da $D$ innerhalb des Abhängigkeitsgraphen $G_1$ reloziert wird, findet die $<$-Relation Anwendung, und Funktionen aus dem Abhängigkeitsgraphen $G_2$ liefern unerwartete Ergebnisse[8].
  2. Objektmodul $D$ benutzt eine durch $A$ definierte Funktion `f()`, im gesamten Abhängigkeitsgraphen $G_2$ wird jedoch keine solche Funktion definiert. Wird der Abhängigkeitsgraph $G_1$ gelöscht, so bleibt $D$ als Bestandteil des Abhängigkeitsgraphen $G_2$ erhalten, seine Referenzen zu `f()` sind jedoch fortan undefiniert.

  Zur Vermeidung der hier genannten Probleme sollten Nutzungsbeziehungen in der oben aufgeführten Form vermieden werden. Stattdessen empfiehlt sich eine konsequente „Top-Down-Benutzung“ von Symboldefinitionen innerhalb eines Abhängigkeitsgraphen.

- **Globale Symbolsuche**
  Durch die globale Symbolsuche kann bewirkt werden, daß ein Objektmodul eines Abhängigkeitsgraphen $G_i$ Symboldefinitionen eines anderen, bezüglich der enthaltenen Objektmodule disjunkten Abhängigkeitsgraphen $G_j$ verwendet. Diese Tatsache spiegelt sich jedoch nicht in der Abhängigkeitsstruktur der Objektmodule wieder und bewirkt, daß $G_j$ gelöscht werden kann, obwohl dessen Symbolwerte in $G_i$ noch in Benutzung sind.

- **Symboladressen nach einer *link*- oder *relink*-Operation**
  Wird ein Objektmodul im Zug einer *link*- oder *relink*-Operation ersetzt oder gelöscht (z. B. durch Maßnahmen der Versionsverwaltung oder eine strukturverändernde *relink*-Operation), so verlieren alle mittels der *getsym*-Operation ermittelten Adressen $a_1, a_2, \ldots$ von Symbolwerten dieses Objektmoduls ihre Gültigkeit.

[8] Dieses Beispiel dient nur der Illustration, in der Praxis würde `sort()` mittels eines Parameters ein Zeiger auf `rel()` übergeben.

Zur Lösung dieses Problems sollen drei Möglichkeiten angedeutet werden:

1. Die Benutzung des dynamischen Linkers wird so organisiert, daß nach jeder *link*- oder *relink*-Operation die Adressen $a_1, a_2, \ldots$ mittels der *getsym*-Operation neu bestimmt werden.
2. Die *getsym*-Operation wird nicht als Funktion implementiert, die die Adresse eines Symbolwerts zurückgibt, sondern als Prozedur, die einen Referenzparameter $r$ mit dieser Adresse belegt. Dabei vermerkt *getsym* die Adresse von $r$ zusammen mit dem Namen des angeforderten Symbols in einer speziellen Liste des Abhängigkeitsgraphen. Werden Objektmodule aus dem Graphen ersetzt oder gelöscht, so wird diese Liste durchmustert. Ein durch einen Eintrag der Liste benanntes Symbol wird erneut gesucht und der „ehemalige“ Referenzparameter $r$ mittels der neubestimmten Adresse aktualisiert. Probleme ergeben sich hierbei jedoch, wenn der Wert des Referenzparameters einer anderen Variablen zugewiesen wurde oder wenn der Speicherbereich von $r$ zum Zeitpunkt der *link*- oder *relink*-Operation nicht mehr verfügbar ist (z. B. wenn der Block, der $r$ definiert, bereits wieder verlassen wurde.).
3. Dem Nutzer des dynamischen Linkers wird anstelle der *getsym*-Operation eine Klasse `Symbol` zur Verfügung gestellt, die in folgender Weise zu verwenden ist:

   ```
   Module G = "libA.so";
   Symbol s (G, "foo(void)");

   ( (void(*)()) s.addr())();
   ```

   Ein Objekt `s` dieser Klasse wird erzeugt, indem durch den Konstruktor ein als Parameter anzugebender Abhängigkeitsgraph `G` nach einem ebenfalls als Parameter anzugebenden Symbolnamen durchsucht wird. Der Konstruktor trägt `s` in eine Liste von `Symbol`-Objekten von `G` ein und vermerkt umgekehrt einen Verweis auf `G` in `s`. Werden Objektmodule aus `G` gelöscht oder ersetzt, so wird die Liste der `Symbol`-Objekte durchmustert und aktualisiert. Wird `s` gelöscht, so sorgt der Destruktor der Klasse `Symbol` dafür, daß `s` aus der Liste von `Symbol`-Objekten von `G` entfernt wird. Umgekehrt werden beim Löschen von `G` alle in der Liste enthaltenen `Symbol`-Objekte als ungültig markiert. Bei einer Anweisung der Form

   ```
   Symbol s2 = s;
   ```

   kann durch einen entsprechend gestalteten Konstruktor der Klasse `Symbol` erreicht werden, daß `s2` auf die gleiche Weise wie `s` mit `G` verknüpft wird.

In der vorliegenden Implementation des dynamischen Linkers ergibt sich die Hauptspeicherrepräsentation eines Objektmoduls durch das Lesen der entsprechenden Datei. Hierfür ist die Methode `load()` der Klasse `File` verantwortlich. Diese Methode könnte so modifiziert werden, daß sie den Inhalt eines Objektmoduls aus einer anderen Datenquelle, z. B. einer Netzwerkverbindung, liest. Beim Entwurf des dynamischen Linkers hatte die einfache Implementation und effektive Ausführung der Algorithmen Vorrang vor einem möglichst geringen Hauptspeicherbedarf. Folgende Maßnahmen können zur Verringerung des Hauptspeicherbedarfs dienen:

- **Re-Relokation**

  Zur Wahrung der Idempotenz bei der wiederholten Durchführung von Relokationen wird der ursprüngliche Inhalt *aller* Relokationsfelder in einem eigens dafür reservierten Speicherbereich (Instanzvariable `orgrel` der Klasse `File`) aufbewahrt. Dies ist jedoch nur für Relokationsfelder notwendig, die einen impliziten Zusatz enthalten, der bei der Adreßberechnung im Zug der Relokation verwendet wird. Es kann also Speicherplatz gespart werden, wenn nur der Inhalt dieser Relokationsfelder aufbewahrt wird.

- **Tabelle von Prototypnamen**

  Die vorliegende Implementation des dynamischen Linkers konvertiert beim Laden eines Objektmoduls jeden in dessen Symboltabelle enthaltenen dekorierten Symbolnamen durch eine

Demangling-Funktion in den entsprechenden Prototypnamen und speichert diesen in einer Tabelle (Instanzvariable `dmgsnm` der Klasse `File`) ab. Dabei werden auch nichtdekorierte Symbolnamen in diese Tabelle übernommen, wodurch die Symbolsuche einzig über dieser Tabelle operieren kann.

Um Speicherplatz zu sparen ist es denkbar, nur Prototypnamen in diese Tabelle zu übernehmen. Weiterhin ist es vorstellbar, die Demangling-Funktion direkt bei der Symbolsuche zu verwenden. Hierdurch würde die Tabelle von Prototypnamen gänzlich überflüssig.

# Kapitel 7

# Zusammenfassung und Ausblick

Das dynamische Linken ist ein Mechanismus, der die Adaption von Software-Systemen zur Laufzeit erlaubt. Dies wird durch das dynamische Laden und Entfernen von Programmkomponenten (Objektmodulen) ermöglicht. Dynamisches Linken ist durch die Auflösung symbolischer Referenzen zwischen Objektmodulen während des Programmstarts bzw. während der Programmausführung gekennzeichnet. Objektmodule sind Datenstrukturen, deren Aufbau durch das Objektdateiformat geregelt wird. Im Rahmen der vorliegenden Arbeit konnte festgestellt werden, daß sich das Objektdateiformat ELF gut als Basis des dynamischen Linkens eignet. Die Benutzung gemeinsamen Speichers und der Erhalt symbolischer Informationen zur Laufzeit bewirkt, daß der Programmcode von Objektmodulen an den Code mehrerer Anwendungsprogramme gebunden und somit geteilt benutzt werden kann. Durch die geteilte Benutzung von Objektmodulen verringert sich die Größe ausführbarer Dateien und der von ihnen während der Abarbeitung effektiv belegte Hauptspeicherplatz. Mit positionsunabhängigem Code wurde ein Methode vorgestellt, durch die geteilt benutzte Objektmodule frei im Adreßraum eines Prozesses angeordnet werden können. Objektdateien einer dynamisch gelinkten Anwendung stellen eine logische Einheit dar, die vom Laufzeit-Linker zu einer physischen Einheit (Process Image) verbunden werden. Die logische Einheit ergibt sich aus der Referenzierung (Benennung) benötigter Objektmodule. Dies führt zur Darstellung von Abhängigkeitsgraphen. Mit der vorliegenden Implementation konnte gezeigt werden, daß sich die Funktionen eines Werkzeugs zum dynamischen Linken als Operationen über Abhängigkeitsgraphen darstellen lassen. Das dynamische Linken und die damit erreichte Flexibilität im Aufbau dynamisch gelinkter Anwendungen macht Maßnahmen zur Gewährleistung der Typsicherheit und zur Verwaltung verschiedener Versionen eines Objektmoduls notwendig. Der implementierte Prototyp eines dynamischen Linkers zeigt, wie sich Typsicherheit auf die Übereinstimmung von Symbolnamen zurückführen läßt und wie durch die Verwaltung von Major- und Minor-Versionsnummern Kompatibilität und Aktualität bei der Verwendung von Objektmodulen erreicht werden kann.

Der dynamische Linker kann zur Erlangung der dynamischen Adaptierbarkeit eines Betriebssystems eingesetzt werden. Hierfür ist im Rahmen weiterführender Arbeiten eine Integration des Linkers in einen Betriebssystem-Kernel vorzunehmen. Da der Kernel hierbei an die Stelle der Linker-Applikation tritt, ist insbesondere die Symbolsuche in der Linker-Applikation (s. Abschn. 6.1.8, S. 60) durch eine Symbolsuche im Kernel zu ersetzen. Dies bedingt, daß eine Kernel-Symboltabelle verfügbar gemacht werden muß. Im Anhang werden hierfür konkrete Hinweise angegeben (s. Anh. A.4, S. 95).

Existierende Objektdateiformate bieten i. allg. nur unzureichende Möglichkeiten zur Beschreibung von Typinformationen. In zukünftigen Untersuchungen sollten Wege analysiert werden, wie Objektmodule um Typinformationen ergänzt werden können. Die Einführung von Typdeskriptoren wie im Fall des HP/UX-a.out-Formats wäre hier vorstellbar, bedingt jedoch eine Modifikation von Compilern oder die Schaffung zusätzlicher Werkzeuge. Die Flexibilität des ELF-Objektdateiformats erlaubt in diesem Zusammenhang die Definition neuer Sektionen, so z. B. zur Unterbringung entsprechender Deskriptortabellen. Einträge einer solchen Tabelle könnten die Typen globaler Variablen bzw. der Rückgabewerte und Parameter von Funktionen beschreiben. Es besteht die Möglichkeit, bislang ungenutzte Attribute eines Symboltabelleneintrags zum Verweis auf einen solchen Typdeskriptor zu benutzen. Referenzen zwischen Typdeskriptoren würden die Darstellung von strukturierten Daten-

typen und von Klassenhierarchien ermöglichen. Für Anregungen in diesem Zusammenhang sei auf Arbeiten wie [Interrante90] oder [Wallace] verwiesen.

Der Link-Editor überprüft während der Erzeugung eines dynamisch gelinkten Programms die Auflösbarkeit symbolischer Referenzen und verfügt daher über Informationen, welches Objektmodul ein referenziertes Symbol definiert. Diese Informationen werden jedoch wieder verworfen und der Laufzeit-Linker muß ggf. erneut eine Menge von Objektmodulen nach einer Symboldefinition durchsuchen. Es ist denkbar, daß bereits während der Erzeugung eines Objektmoduls für dessen Symboldeklarationen vermerkt wird, welches Objektmodul die entsprechende Definition enthält. Unter gewissen Voraussetzungen könnten für diese Vermerke Referenzen auf die Einträge der Abhängigkeitsliste eines Objektmoduls verwendet werden.

Bei der objektorientierten Realisierung eines dynamischen Linkers bietet sich die Repräsentation von Objektmodulen durch Instanzen entsprechender Klassen an (z. B. der Klasse `File` im Fall der vorliegenden Implementation). Um dem dynamischen Linker die Verarbeitung einer Vielzahl von Objektdateiformaten und -typen zu ermöglichen, könnte hierfür eine Klassenhierarchie eingeführt werden. Von einer abstrakten Basisklasse, die alle grundlegenden Eigenschaften eines Objektmoduls repräsentiert, wären in einem solchen Fall weitere Klassen zur konkreten Darstellung von Objektdateitypen verschiedener Objektdateiformate abzuleiten (s. Abb. 7.1).

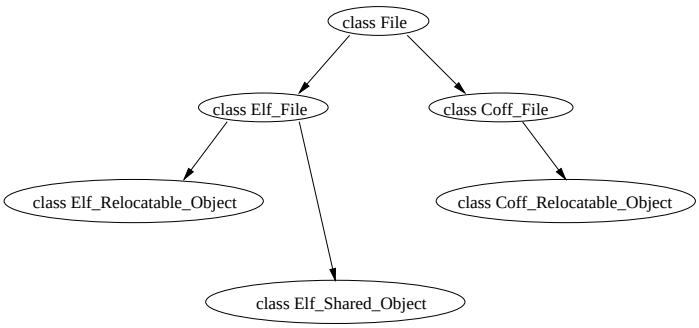


Abbildung 7.1: Beispiel einer Klassenhierarchie für Objektdateitypen

Es verbleibt festzustellen, daß mit dem Einsatz des dynamischen Linkens ein vielversprechender Weg beschritten wird, und es ist zu hoffen, daß die Forschungen auf diesem Gebiet durch weitere Arbeiten vorangetrieben werden.

# Literaturverzeichnis


[ATT91a] AT&T, UNIX System Laboratories, Inc.: *UNIX System V: Application Binary Interface: Intel i860 Processor Supplement.*
Prentice Hall, 1991.

[ATT91b] AT&T, UNIX System Laboratories, Inc.: *UNIX System V/386 Release 4: Leitfaden für die Systemneuerungen.*
Prentice Hall, 1991.

[ATT91c] AT&T, UNIX System Laboratories, Inc.: *UNIX System V Release 4: Leitfaden für Programmierer: ANSI-C und Programmierwerkzeuge.*
Prentice Hall, 1991.

[Claßen90] Claßen, Ludwig: *Programmierhandbuch 80386/80486.*
Verlag Technik, Berlin, 1990.

[Dilger93] Dilger, Werner: *Einführung in die künstliche Intelligenz.*
Vorlesungsskript, Technische Universität Chemnitz-Zwickau, Fakultät für Informatik, Professur Künstliche Intelligenz, 1993.

[Engel95] Engel, David: *Linux shared, dynamic linker and utilities - Version 1.7.14.*
Quelltext, 1995.
`ftp:/tsx-11.mit.edu/pub/linux/packages/GCC/ld.so-1.7.14.tar.gz`

[Gircys88] Gircys, Gintaras R.: *Understanding and Using COFF.*
O'Reilly & Associates Inc., Sebastopol, 1988.

[Ho90] Ho, W. Wilson; Olson, Ronald A.: *An Approach to Genuine Dynamic Linking.*
Technischer Bericht, Division of Computer Science, University of California, Davis, 1990.
`http://nswt.tuwien.ac.at:8000/htdocs/gnudoc/dld/SPE.ps`

[Ho91] Ho, W. Wilson: *A Dynamic Link/Unlink Editor - Version 3.2.3.*
Technischer Bericht, Division of Computer Science, University of California, Davis, 1991.

[Interrante90] Interrante, John A.; Linton, Mark A.: *Runtime Access to Type Information in C++.*
Technischer Bericht (CS-TR-90-418), Stanford University, 1990.

[Kalfa88] Kalfa, Winfried: *Betriebssysteme.*
Akademie-Verlag, Berlin, 1988.

[Kalfa96] Graupner, Sven; Kalfa, Winfried (Hrsg.); Schubert, Frank; Vogel, Ronny; Werner, Jörg; Wohlrab, Lutz: *Dynamische Adaption in Betriebssystemen - Das CHEOPS-Projekt.*
Chemnitzer Informatik-Berichte (CSR-96-03), Technische Universität Chemnitz-Zwickau, Fakultät für Informatik, Professur Betriebssysteme, 1996.
`http://www.tu-chemnitz.de/informatik/osg/papers/96/CSR-96-03/itti/itti.html`

[Kempf92] Kempf, J.; Kessler, P. B.: *Cross-Address Space Dynamic Linking.*
Technischer Bericht (TR-92-2), Sun Microsystems Laboratories Inc., 1992.

[Lu95] Lu, Hongjiu: *ELF: From The Programmer's Perspective.*
Online-Dokument, NYNEX Science & Technology Inc., 1995.
`ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz`

[Schubert96] Schubert, Frank: *Dynamische Adaptierbarkeit in Betriebssystemen auf Basis objektorientierter Methoden.*
Technische Universität Chemnitz-Zwickau , Fakultät für Informatik, Professur Betriebssysteme, In *[Kalfa96]*, 1996.
`http://www.tu-chemnitz.de/informatik/osg/papers/96/CSR-96-03/coc/coc.html`

[Ellis91] Ellis, Margaret A.; Stroustrup, Bjarne: *The Annotated C++ Reference Manual.*
Addison Wesley, 1991.

[Sun90] Sun Microsystems: *Programming Utilities & Libraries (Solaris 1.1.1 Answerbook, Administrator's Set).*
Sun Microsystems Inc., 1990.

[Sun91] SunPro: *SPARCompilers C 2.0.1 Programmer's Guide (SPARCworks and SPARCompilers 2.0 Answerbook).*
Sun Microsystems Inc., 1991.

[Sun92] SunSoft: *SPARCompiler C++ 4.0.1 Name Mangling Document.*
Online-Dokument der SPARCompiler C++-Distribution, Sun Microsystems Inc., 1992.

[Sun93a] SunSoft: *SunOS 5.3 Linker and Libraries Manual (Solaris 2.3 Software Developer AnswerBook).*
Sun Microsystems Inc., 1993.

[Sun93b] SunSoft: *Assembly Language Reference Manual for SPARC (Solaris 2.3 Software Developer AnswerBook).*
Sun Microsystems Inc., 1993.

[TIS94] Tool Interface Standards (TIS): *Executable and Linkable Format (ELF).*
TIS, Portable Formats Specification, Version 1.1, 1994.
`ftp://tsx-11.mit.edu/pub/linux/packages/GCC/ELF.doc.tar.gz`

[Wallace] Wallace, David Vinayak: *Runtime Type Support in C and C++.*
Technischer Bericht, Cygnus Online Library.
`http://www.cygnus.com/library/ctr/tdesc.ps`

# Glossar

**Abhängigkeit**
: Wird zur Erstellung eines dynamisch zu linkenden Objektmoduls (geteilt benutzbare Objektdatei, dynamisch gelinktes Programm) ein weiteres dynamisch zu linkendes Objektmodul (geteilt benutzbare Objektdatei) verwendet, so wird im zu erstellenden Objektmodul eine Referenz auf das verwendete Objektmodul vermerkt. Diese Referenz wird als *Abhängigkeit* und das referenzierte Modul als *benötigtes Objektmodul* des zu erstellenden Objektmoduls bezeichnet.

**Abhängigkeitsliste**
: Alle Abhängigkeiten eines Objektmoduls formen dessen Abhängigkeitsliste.

**Abhängigkeitsgraph**
: Betrachtet man Objektmodule als Knoten und verbindet diese mittels gerichteter Kanten mit den durch die Abhängigkeitsliste bezeichneten Objektmodulen, so ergibt sich der *Abhängigkeitsgraph*.

**Adresse, absolute**
: Eine *absolute Adresse* bezeichnet eine Position im Adreßraum eines Prozesses.

**Adresse, relative**
: Als *relative Adresse* wird eine Hauptspeicherposition bezüglich einer Basisadresse (z. B. der eines Objektmoduls) bezeichnet.

**Anwendung, dynamisch gelinkte**
: Als *dynamisch gelinkte Anwendung* wird die Gesamtheit aller zum Abhängigkeitsgraphen eines dynamisch gelinkten Programms gehörenden Objektmodule bezeichnet.

**Archiv**
: Ein *Archiv* (`*.a`) ist eine Datei, in der Kopien mehrerer relozierbarer Objektdateien enthalten sind. Archive verfügen über eine Archiv-Symboltabelle und werden vom Link-Editor als Eingabe verarbeitet.

**Basisadresse**
: Die *Basisadresse* ist eine absolute Adresse und bezeichnet den Beginn des von einem Objektmodul belegten Speicherbereichs. Die Basisadresse ist der Bezugspunkt relativer Adreßangaben (Symbolwerte, Relokationsfelder) innerhalb eines Objektmoduls.

**`.bss`-Sektion**
: Die `.bss`-*Sektion* („*Block Started by Symbol*“) bezeichnet den Abschnitt eines ELF-Objektmoduls, der zur Aufbewahrung nichtinitialisierter Variablen dient. Diese Sektion belegt keinen Platz in der Datei. Für diese Sektion wird zur Laufzeit Speicher gemäß der vom SHT-Eintrag angegebene Größe allokiert.

**Datei, ausführbare**
: Eine *ausführbare Datei* ist eine Objektdatei, die einen Eintrittspunkt definiert und vom Programmlader gestartet werden kann.

**Eintrittspunkt**
Der *Eintrittspunkt* bezeichnet die absolute Adresse, an der die Abarbeitung des Programmcodes einer ausführbaren Datei oder einer Funktion beginnt.

**Interposition**
Die Interposition ist eine Resolutionsregel, nach der die zuerst aufgefundene von mehreren gleichnamigen Symboldefinitionen verwendet wird.

**Laden**
Als *Laden* wird der Vorgang bezeichnet, durch den der Inhalt eines Objektmoduls in den Adreßraum eines Prozesses eingebracht wird. Hierfür wird i. allg. der Inhalt einer Objektdatei gelesen, bei geteilter Benutzung von Objektmodulen kann „Laden“ aber auch das Einblenden von Abschnitten gemeinsamen Speichers bedeuten.

**Laufzeit-Linker**
Der *Laufzeit-Linker* ist ein Programm(-modul), das während der Laufzeit einer Anwendung Objektmodule lädt oder entfernt und symbolische Referenzen zwischen diesen Objektmodulen und der Anwendung auflöst. Der Laufzeit-Linker wird auch als *dynamischer Linker* bezeichnet.

**Linken, dynamisches**
*Dynamisches Linken* ist der Oberbegriff für Methoden im Zusammenhang mit dem Laden und Entfernen von Objektmodulen sowie der Auflösung symbolischer Referenzen während der Laufzeit eines Programms.

**Linken, statisches**
*Statisches Linken* bezeichnet die Konkatenation relozierbarer Objektdateien zu einer Ausgabedatei (statisch gelinktes Programm, relozierbare Objektdatei).

**Link-Editor**
Der *Link-Editor* ist ein (Kommandozeilen-) Programm (z. B. `ld(1)`), das relozierbare bzw. geteilt benutzbare Objektdateien sowie Archive als Eingabe verarbeitet und eine relozierbare bzw. geteilt benutzbare Objektdatei oder ein statisch bzw. dynamisch gelinktes Programm als Ausgabe erzeugt.

**Objektdatei**
Eine *Objektdatei* beinhaltet die (oder Teile der) Maschinensprache-Repräsentation eines Programms. Objektdateien werden in statisch bzw. dynamisch gelinkte Programme und relozierbare bzw. geteilt benutzbare Objektdateien unterschieden.

**Objektdatei, geteilt benutzbare**
*Geteilt benutzbare Objektdateien* (*shared objects*, `*.so`) stellen die Eingabe des dynamischen Linkers dar. Sie sind mit speziellen Informationen für das dynamische Linken ausgestattet. Ihr Programmcode kann von mehreren Prozessen geteilt benutzt werden.

**Objektdatei, relozierbare**
*Relozierbare Objektdateien* (*relocatable objects*, `*.o`) stellen die Ausgabe des Übersetzungssystems dar oder werden durch die Konkatenation bereits existierender relozierbarer Objektdateien durch den Link-Editor erzeugt. Relozierbare Objektdateien werden i. allg. beim statischen Linken verwendet.

**Objektmodul**
Der Begriff *Objektmodul* kennzeichnet eine Objektdatei, eine Komponente eines Archivs oder den Inhalt einer Objektdatei, die bereits in den Hauptspeicher eingelesen wurde.

**Programm, dynamisch gelinktes**
Ein *dynamisch gelinktes Programm* (*dynamic executable*) ist eine ausführbare Datei, deren symbolische Referenzen vom Laufzeit-Linker aufgelöst werden müssen. Dafür lädt der Laufzeit-Linker die von einem dynamisch gelinkten Programm referenzierten geteilt benutzbaren Objektdateien.

**Programm, statisch gelinktes**
Ein *statisch gelinktes Programm* (*static executable*) ist eine ausführbare Datei, bei deren Erzeugung alle symbolischen Referenzen durch den Link-Editor aufgelöst wurden.

**Programmlader**
Unter dem *Programmlader* ist der Systemruf zu verstehen, der für das Laden und den Start ausführbarer Dateien verantwortlich ist (z. B. `exec(2)`).

**Referenz, symbolische**
Eine *symbolische Referenz* bezeichnet ein Adreßfeld in einem Maschinenprogramm (Operand eines Maschinenbefehls, Zeiger o.ä.), das im Zug der sog. *Auflösung* mit der Adresse eines Symbolwerts überschrieben werden muß. Symbolische Referenzen werden durch entsprechende Relokations- und Symbolinformationen dargestellt.

**Relokation**
Die *Relokation* ist der Vorgang, durch den absolute Adreßbezüge eines Objektmoduls an dessen Position im Adreßraum eines Prozesses angepaßt (*nichtsymbolische Relokation*) oder mittels der absoluten Adresse eines Symbolwerts modifiziert (*symbolische Relokation*) werden. Relokationen werden durch Relokationsinformationen gesteuert.

**Resolution**
Die *Resolution* stellt die Anwendung eines Satzes von Vorrangregeln dar, durch die die Verwendung einer von mehreren gleichnamigen Symboldefinitionen festgelegt wird.

**Symbol**
Ein Symbol ist eine Datenstruktur innerhalb eines Objektmoduls, die einen definierten oder referenzierten Symbolwert durch einen Namen identifiziert.

**Symboldefinition**
Eine *Symboldefinition* bezeichnet ein Symbol für einen innerhalb eines Objektmoduls definierten Symbolwert.

**Symboldeklaration**
Eine *Symboldeklaration* bezeichnet ein Symbol für einen innerhalb eines Objektmoduls benötigten, nicht aber innerhalb dieses Moduls definierten Symbolwert. Symboldeklarationen werden auch als *undefinierte Symbole* bezeichnet.

**Symbolwert**
Der *Symbolwert* ist die Maschinensprache-Repräsentation von Elementen der Programmiersprache, so z. B. von globalen Variablen oder Funktionen.

**Übersetzungssystem**
Das Übersetzungssystem erzeugt die Maschinensprache-Repräsentation eines Programms in Form einer (relozierbaren) Objektdatei. Es besteht aus Präprozessor (optional), Compiler und Assembler (optional).

# Abbildungsverzeichnis


2.1 Einsatz des Linkers bei der Programmentwicklung . . . . . 5
2.2 Situation vor Durchführung einer symbolischen Relokation . . . . . 12
2.3 Situation nach Durchführung einer symbolischen Relokation . . . . . 13

3.1 Beispiel eines Abhängigkeitsgraphen . . . . . 17
3.2 Geteilte Benutzung bei absoluter Adressierung . . . . . 18
3.3 Geteilte Benutzung bei relativer Adressierung . . . . . 19
3.4 Geteilte Benutzung bei privater Modifikation . . . . . 20
3.5 Abhängigkeiten von Symbolwerten . . . . . 24

4.1 Aufbau einer ELF-Objektdatei . . . . . 25
4.2 Textsegment . . . . . 30
4.3 Datensegment . . . . . 30

5.1 Link-Editor: statisches Linken . . . . . 37
5.2 Link-Editor: dynamisches Linken . . . . . 38
5.3 Laden eines Segments . . . . . 42
5.4 Abarbeitungsreihenfolge des Initialisierungs- und Terminierungscodes . . . . . 44
5.5 GOT-Zugriff . . . . . 47
5.6 Initialer Aufruf eines PLT-Eintrags . . . . . 49
5.7 Nachfolgende Aufrufe eines PLT-Eintrags . . . . . 50
5.8 Versionsverwaltung unter ELF . . . . . 52

6.1 Abhängigkeitsgraph mit Referenzzählern . . . . . 54
6.2 Objektmodule als Elemente mehrerer Abhängigkeitsgraphen . . . . . 55
6.3 Einfache *relink*-Operation . . . . . 56
6.4 Strukturverändernde *relink*-Operation (1) . . . . . 57
6.5 Strukturverändernde *relink*-Operation (2) . . . . . 57
6.6 Klassenhierarchie und Abhängigkeitsgraph . . . . . 57
6.7 Module Chain . . . . . 63
6.8 Instanzvariablen der Klasse `File` . . . . . 64
6.9 File Chain . . . . . 65
6.10 Verwendung von `Filelist`-Objekten . . . . . 65
6.11 Relokation und Symbolsuche . . . . . 69

7.1 Beispiel einer Klassenhierarchie für Objektdateitypen . . . . . 76

A.1 Quelltextstruktur des dynamischen Linkers . . . . . 92

A.2 Beispieldateien (1) . . . . . 97
A.3 Beispieldateien (2) . . . . . 99
A.4 Beispieldateien (3) . . . . . 99
A.5 Beispieldateien (4) . . . . . 100
A.6 Beispieldateien (5) . . . . . 100
A.7 Beispieldateien (6) . . . . . 100
A.8 Beispieldateien (7) . . . . . 101

# Tabellenverzeichnis


4.1 Sektionen einer ELF-Datei . . . . . 29
4.2 Größen zur Berechnung des Relokationswerts . . . . . 33

6.1 Versionsverwaltung während der *link*-Operation . . . . . 59
6.2 Versionsverwaltung während der *relink*-Operation . . . . . 59

A.1 Quelltextdateien des dynamischen Linkers . . . . . 91
A.2 Parameter für die Erzeugung eines Abhängigkeitsgraphen . . . . . 93
A.3 Parameter für die Symbolsuche . . . . . 93
A.4 Fehlernummern des dynamischen Linkers . . . . . 94
A.5 Parameter für die Anzeige von Statusinformationen . . . . . 94
A.6 Parameter für die *relink*-Operation . . . . . 94
A.7 Systemschnittstelle des dynamischen Linkers . . . . . 95

B.1 Initialisierungs- und Terminierungscode mit dem Sun-C++-Compiler . . . . . 105
B.2 Initialisierungs- und Terminierungscode mit dem GNU-C++-Compiler . . . . . 108

# Verzeichnis der Code-Beispiele


2.1 Verwendung schwach gebundener Symbole . . . . . 11
5.1 `libpic.c` . . . . . 47
5.2 `libpic.so` (1) . . . . . 48
5.3 `libpic.so` (2) . . . . . 50
5.4 `libpic.so` (3) . . . . . 51
A.1 Symbolsuche im Linux-Kernel . . . . . 96
A.2 Beispiel für die Verwendung des dynamischen Linkers . . . . . 98
B.1 `itc.C` . . . . . 104
B.2 `.init`- und `.fini`-Sektionen mit dem Sun-C++-Compiler . . . . . 104
B.3 `.ctors`- und `.dtors`-Sektionen . . . . . 106

# Verzeichnis der Algorithmen

6.1 Breite-zuerst-Durchmusterung eines Abhängigkeitsgraphen . . . . . 66
6.2 Tiefe-zuerst-Durchmusterung eines Abhängigkeitsgraphen . . . . . 67
6.3 Erzeugung eines Abhängigkeitsgraphen . . . . . 67
6.4 Dekrementierung von Referenzzählern . . . . . 68
6.5 Symbolsuche . . . . . 70
6.6 Aktualisierung von Abhängigkeitsgraphen nach einer *relink*-Operation . . . . . 71

# Anhang A

# Implementationsbeschreibung

## A.1 Verzeichnis- und Dateistruktur

Der implementierte dynamische Linker wird in Form der gepackten Datei `odl-`$x$`.`$y$`.taz`[1] zur Verfügung gestellt. Nach dem Entpacken der Datei mittels

```
tar -xzf odl-x.y.taz
```

liegt das Verzeichnis `odl-`$x$`.`$y$ vor, das die Unterverzeichnisse `odl/`, `demo/` und `tools/` enthält. Jedes dieser vier Verzeichnisse enthält eine Datei `README` mit einer Beschreibung des Verzeichnisinhalts. Darüber hinaus ist in jedem der Verzeichnisse eine Datei `Makefile` enthalten. Die Eingabe von `make` in einem der Unterverzeichnisse bewirkt die Erzeugung der darin enthaltenen Programme und Objektdateien (s. u.), durch `make` im Verzeichnis `odl-`$x$`.`$y$ werden die Programme und Objektdateien aller Unterverzeichnisse erzeugt.

Im Unterverzeichnis `odl/` befinden sich die Quelltextdateien des dynamischen Linkers (s. Abb. A.1). Die `*.C`- bzw. `*.c`-Dateien werden beim Aufruf von `make` separat übersetzt. Die dabei erzeugten relozierbaren Objektdateien werden anschließend in das Archiv `libodl.a` kopiert. Tabelle A.1 beschreibt die Quelltextdateien und ihren Inhalt.

| Datei | Beschreibung |
|---|---|
| `odl.H` | die mittels `#include` in die Linker-Applikation einzuschließende Header-Datei |
| `module.(H\|C)` | Klasse `Module` |
| `file.(H\|C)` | Klasse `File` |
| `filelist.(H\|C)` | Klasse `Filelist` |
| `ksym.(H\|C)` | Funktionen zur Symbolsuche in der Linker-Applikation |
| `misc.(H\|C)` | verschiedene Hilfsfunktionen und Makros |
| `demangle.h` `cplus-dem.c` | Demangling-Funktion (diese Dateien entstammen der GNU-C/C++-Distribution) |
| `verbose.h` | Namen verschiedener ELF-Konstanten (Verwendung bei gesetzter Option `DEBUG`, um Fehlernachrichten „lesbarer“ zu gestalten) |

Tabelle A.1: Quelltextdateien des dynamischen Linkers

Im Verzeichnis `demo/` sind verschiedene Anwendungsbeispiele (s. Abschn. A.5, S. 97) enthalten. Das Verzeichnis `tools/` beinhaltet die (Quelltexte für die) folgenden Dateien:

[1] Die Versionsnummern $x$ und $y$ sind Gegenstand von Veränderungen.

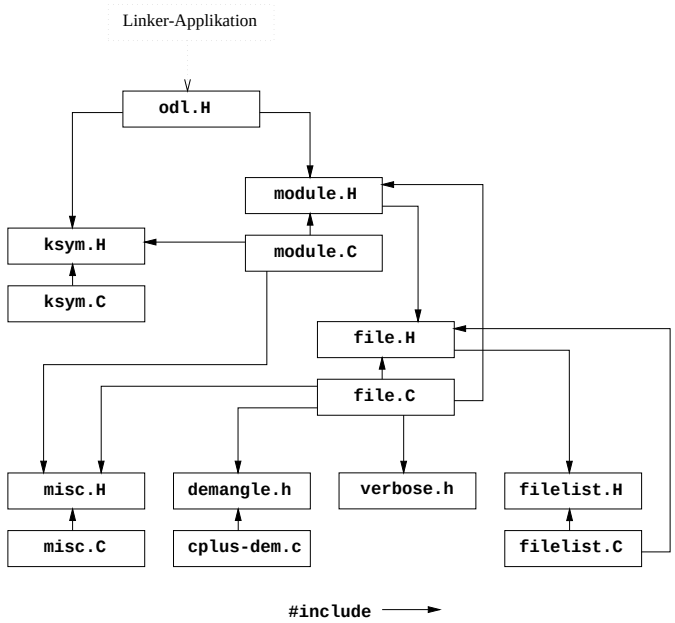


Abbildung A.1: Quelltextstruktur des dynamischen Linkers

- `mkshobj`
  Dieses Shell-Skript erzeugt eine geteilt benutzbare Objektdatei so, daß das Ende des Textsegments und der Anfang des Datensegments bündig aneinanderstoßen. Hierdurch wird die sonst übliche Lücke von der Größe einer Speicherseite (s. Abschn. 5.3, S. 41) eliminiert und damit die feingranulare Adaptierbarkeit einer dynamisch gelinkten Anwendung verbessert. Durch `mkshobj` wird eine geteilt benutzbare Objektdatei durch entsprechende Compiler-Aufrufe zweimal erzeugt. Nach dem ersten Compiler-Aufruf wird mittels `eots` (s. u.) die letzte (relative) Adresse des Textsegments bestimmt. Beim zweiten Compiler-Aufruf wird der Link-Editor angewiesen, den Beginn des Datensegments unmittelbar auf diese Adresse folgen zu lassen.

- `eots`
  Das Programm `eots` (*E*nd *O*f *T*ext *S*egment) ermittelt die letzte Adresse des Textsegments einer geteilt benutzbaren Objektdatei und schreibt diese auf die Standard-Ausgabe.

- `edump`
  Unter Linux steht das Programm `objdump(1)` zur Anzeige des Inhalts einer Objektdatei zur Verfügung. Durch `objdump(1)` können jedoch nicht alle ELF-spezifischen Informationen dargestellt werden. Das Programm `edump` (*E*LF *dump*) ermöglicht die Ausgabe der Informationen, die durch `objdump(1)` nicht verfügbar gemacht werden können. Dazu zählen z.B. das Dynamic-Segment, die Symboltabelle für das dynamische Linken, der ELF-Header oder die PHT (s. `edump -h`).

## A.2 Parameter der Nutzerschnittstelle

Die folgende Aufzählung gibt an, welche Parameterkonstanten zur Steuerung der Funktionen des dynamischen Linkers durch den Nutzer verwendet werden können:

1. **Erzeugung von Abhängigkeitsgraphen**
   Durch den Parameter `mode` des Konstruktors

```
Module(char *name, unsigned int mode = MOD_LOCAL);
```

der Klasse `Module` kann der zu erzeugende Abhängigkeitsgraph zur globalen Symbolsuche freigegeben werden. Der Parameter kann mit den in Tabelle A.2 dargestellten Werten belegt werden. Wird er nicht angegeben, so wird der Abhängigkeitsgraph standardmäßig nicht zur globalen Symbolsuche freigegeben.

| Parameter | Bedeutung |
|---|---|
| `MOD_LOCAL` | Der Abhängigkeitsgraph wird nicht für die globale Symbolsuche freigegeben. |
| `MOD_GLOBAL` | Der Abhängigkeitsgraph wird für die globale Symbolsuche freigegeben. |

Tabelle A.2: Parameter für die Erzeugung eines Abhängigkeitsgraphen

2. **Symbolsuche**
   Durch den Parameter `mode` der Methode

```
unsigned int getsym(char *name, int mode = SNM_PROTO);
```

   der Klasse `Module` kann festgelegt werden, ob es sich bei dem durch den Parameter `name` angegebenen Symbolnamen um einen Prototypnamen oder einen dekorierten Symbolnamen handelt. Damit wird dem Nutzer die Möglichkeit eingeräumt, unabhängig von der verwendeten Demangling-Funktion nach Symbolen zu suchen. Der Parameter kann mit den in Tabelle A.3 dargestellten Werten belegt werden. Wird er nicht angegeben, so wird `name` standardmäßig als Prototypname behandelt.

| Parameter | Bedeutung |
|---|---|
| `SNM_PROTO` | Der angegebene Symbolname ist als Prototypname zu behandeln. |
| `SNM_NOPROTO` | Symboltabellen werden direkt nach dem angegebenen Symbolnamen durchsucht. |

Tabelle A.3: Parameter für die Symbolsuche

3. **Fehlernummern**
   Tritt während des Ladens oder der Relokation von Objektmodulen ein Fehler auf, so kann mit der Methode

```
int geterrno();
```

   der Klasse `Module` eine Fehlernummer abgefragt werden. Tabelle A.4 stellt die möglichen Fehlernummern und ihre Bedeutung dar.

4. **Anzeige von Statusinformationen**
   Durch den Parameter `mode` der Methode

```
void dispinfo(int mode = DSP_FILES | DSP_SCOPE);
```

   der Klasse `Module` kann festgelegt werden, welche Informationen über den Status des dynamischen Linkers anzuzeigen sind. Tabelle A.5 stellt die möglichen Anzeigemodi dar. Wird der Parameter nicht angegeben, so wird standardmäßig die Liste aller geladenen Objektmodule sowie deren Anordnung innerhalb der Abhängigkeitsgraphen angezeigt.

| Fehlernummer | Bedeutung |
|---|---|
| DLE_NO_ERR | Es ist kein Fehler aufgetreten. |
| DLE_NOFILE | Eine erforderliche Datei kann nicht gefunden werden. |
| DLE_NO_HDR | Der ELF-Header einer Datei kann nicht gelesen werden. |
| DLE_NOTELF | Das Format einer Datei ist nicht ELF. |
| DLE_WRARCH | Der Inhalt einer Datei bezieht sich auf eine andere Rechnerplattform. |
| DLE_WRTYPE | Eine Datei ist nicht vom Typ einer geteilt benutzbaren Objektdatei. |
| DLE_NO_SYM | Eine symbolische Referenz kann nicht aufgelöst werden. |
| DLE_WR_REL | Ein unerwarteter Relokationstyp liegt vor. |

Tabelle A.4: Fehlernummern des dynamischen Linkers

| Ausgabemodus | Bedeutung |
|---|---|
| DSP_FILES | Anzeige der Liste geladener Objektmodule |
| DSP_SCOPE | Anzeige der Abhängigkeitsgraphen |
| DSP_USAGE | Anzeige der effektiven Nutzungsstruktur (Liste effektiv nutzender und effektiv genutzter Objektmodule) |

Tabelle A.5: Parameter für die Anzeige von Statusinformationen

Um mehrere Ausgaben gleichzeitig zu erhalten, können die `DSP_*`-Konstanten bitweise *ODER*-verknüpft werden.

5. **Relinken**
   Durch den Parameter `mode` der Methode

```
void relink(int mode = REL_ALL);
```

   der Klasse `Module` kann festgelegt werden, ob alle Objektmodule eines Abhängigkeitsgraphen ausgetauscht werden sollen oder nur das Objektmodul, welches die Wurzel des Abhängigkeitsgraphen darstellt. Der Parameter kann mit den in Tabelle A.6 dargestellten Werten belegt werden. Wird er nicht angegeben, so wird standardmäßig der gesamte Abhängigkeitsgraph ausgetauscht.

| Parameter | Bedeutung |
|---|---|
| REL_TOP | Das Objektmodul, das die Wurzel des Abhängigkeitsgraphen darstellt, wird ausgetauscht. |
| REL_ALL | Alle Objektmodule des Abhängigkeitsgraphen werden ausgetauscht. |

Tabelle A.6: Parameter für die *relink*-Operation

## A.3 Systemschnittstelle des dynamischen Linkers

Durch den dynamischen Linker werden die in Tabelle A.7 dargestellten Bibliotheksfunktionen benutzt. Umgebungsvariablen werden durch den dynamischen Linker nicht verwendet.

| Funktion | Verwendung |
|---|---|
| *Dateizugriff* | |
| `open(2)` | Laden von Objektdateien und der Symboltabelle der Linker-Applikation |
| `read(2)` | |
| `lseek(2)` | |
| `close(2)` | |
| *Speicherverwaltung* | |
| `malloc(3)` | impliziter Aufruf durch `strdup(3)` |
| `free(3)` | Freigabe des durch `strdup(3)` allokierten Speichers |
| C++-Operator `new` | jede sonstige Allokation und Freigabe von Speicher |
| C++-Operator `delete` | |
| *Textausgabe* | |
| `printf(3)` | Ausgabe von Statusinformationen |
| `fprintf(3)` | Ausgabe von Fehlerinformationen (nur bei definiertem Makro `DEBUG`) |
| *Zeichenkettenmanipulation* | |
| `strchr(3)` | Analyse und Manipulation von Pfad-, Datei- und Symbolnamen |
| `strlen(3)` | |
| `strcmp(3)` | |
| `strcpy(3)` | |
| `strcat(3)` | |
| `strdup(3)` | |
| `strtok(3)` | |
| *Sonstiges* | |
| `atoi(3)` | Extraktion der Versionsnummern aus Dateinamen |
| `isdigit(3)` | |
| `isalnum(3)` | Komprimierung von Symbolnamen |
| `isspace(3)` | |

Tabelle A.7: Systemschnittstelle des dynamischen Linkers

Das Demangling-Modul `cplus-dem.c` verwendet darüber hinaus folgende Bibliotheksfunktionen:

1. `realloc(3)`
2. `memcmp(3)`
3. `memset(3)`
4. `strncmp(3)`
5. `strncat(3)`
6. `strspn(3)`
7. `strcspn(3)`
8. `strpbrk(3)`

## A.4 Hinweise für die Kernel-Migration

Bei der Integration des dynamischen Linkers in einen Betriebssystem-Kernel müssen die in Abschn. A.3 (S. 94) dargestellten Bibliotheksfunktionen durch entsprechende Kernel-Funktionen ersetzt werden. Im Zug der Migration übernimmt der Kernel die Rolle der Linker-Applikation. Dabei muß insbesondere die Symbolsuche in der Linker-Applikation (s. Abschn. 6.1.8, S. 6.1.8) durch eine Symbolsuche im Kernel ersetzt werden. Hierfür ist eine Kernel-Symboltabelle bereitzustellen.

Im Fall von Linux kann dafür beispielsweise die Kernel-Funktion `get_kernel_syms()` (s. `modules(2)`) verwendet werden. Die Einträge der von dieser Funktion zurückgelieferten Kernel-Symboltabelle sind folgendermaßen definiert:

```
struct kernel_sym{
  unsigned long value;
  char name[SYM_MAX_NAME];
};
```

Die Komponente `value` gibt dabei die Adresse einer Kernel-Funktion an, in `name` ist der Name der Funktion enthalten. Die Verwendung von `get_kernel_syms()` macht eine Veränderung des Quelltextmoduls `ksym.C` und der darin enthaltenen Funktionen `ksyminit()`, `ksymdone()` und `kgetsym()` erforderlich. Code-Beispiel A.1 deutet die notwendigen Modifikationen an.

```
01:   #include <stdlib.h>
02:   #include <linux/module.h>
03:   #include <sys/syscall.h>
04:
05:   #ifndef get_kernel_syms
06:     #define get_kernel_syms(p) syscall(SYS_get_kernel_syms, p)
07:   #endif
08:
09:   struct kernel_sym *ksymtab;
10:   static unsigned    knsym;
11:
12:   void ksyminit(){
13:     knsym = get_kernel_syms(NULL);
14:     ksymtab = (struct kernel_sym*) malloc(knsym * sizeof(struct kernel_sym));
15:     get_kernel_syms(ksymtab);
16:   }
17:
18:   void ksymdone(){
19:     free(ksymtab);
20:   }
21:
22:   void *kgetsym(char *name){
23:     unsigned int i;
24:
25:     for(i = 0; i < knsym; i++){
26:       ...
27:       if(!strcmp(name, ksymtab[i].name)
28:         return((void*) ksymtab[i].value);
29:       ...
30:     }
31:
32:     return(NULL);
33:
34:   }
```

**Code–Beispiel A.1:** Symbolsuche im Linux-Kernel

Durch den ersten Aufruf von `get_kernel_syms()` in der Funktion `ksyminit()` wird die Anzahl der Einträge der Kernel-Symboltabelle ermittelt (Zeile 13). Anhand dieser Anzahl wird Speicherplatz für die Kernel-Symboltabelle allokiert (Zeile 14). Durch einen zweiten Aufruf von `get_kernel_syms()` wird dieser Speicherplatz mit dem Inhalt der Kernel-Symboltabelle belegt (Zeile 15). In der Funktion `ksymdone()` wird der in `ksyminit()` für die Kernel-Symboltabelle allokierte Speicherplatz wieder freigegeben (Zeile 19). In der Funktion `kgetsym()` wird die Kernel-Symboltabelle auf der Suche nach

einer Symboldefinition mit dem Namen `name` durchmustert (Zeilen 25–30). Konnte die entsprechende Symboldefinition gefunden werden, so wird die Adresse des Symbolwerts zurückgegeben (Zeile 28), andernfalls wird der Wert `NULL` zurückgeliefert (Zeile 32).

## A.5 Anwendungsbeispiele

Das Code-Beispiel A.2 demonstriert die Verwendung des dynamischen Linkers. Durch die `#include`-Anweisung in Zeile 02 werden die Schnittstellenvereinbarungen des dynamischen Linkers importiert. In Zeile 09 wird mittels der Funktion `ksyminit()` die Symboltabelle der Linker-Applikation (das aus Code-Beispiel A.2 erzeugte Programm) gelesen. Die Definition des `Modul`-Objekts `A` in Zeile 11 bewirkt, daß die geteilt benutzbare Objektdatei `libA.so` und alle von ihr benötigten Objektmodule (s. u.) geladen und reloziert werden. Nach der Relokation wird der Initialisierungscode dieser Objektmodule ausgeführt. Mit den Zeilen 12 – 15 erfolgt die Behandlung eines möglicherweise in Zeile 11 aufgetretenen Fehlers. Hierfür wird mittels der Methode `geterrno()` eine Fehlernummer abgefragt. Der Rückgabewert `DLE_NO_ERR` zeigt hierbei an, daß kein Fehler vorliegt. Wenn ein Fehler aufgetreten ist, wird mittels der Methode `geterrmsg()` die dazugehörige Fehlermeldung ermittelt und auf der Standard-Fehlerausgabe angezeigt. In Zeile 17 werden mittels der Methode `dispinfo()` Informationen über geladene Objektmodule und die sich daraus ergebenden Abhängigkeitsgraphen angezeigt. Der Aufruf der Methode `getsym()` in Zeile 19 versucht, die Funktion `func(void)` innerhalb des durch Zeile 11 geladenen Abhängigkeitsgraphen zu lokalisieren. Gelingt dies, so wird die Adresse der Funktion dem Zeiger `fp` zugewiesen, andernfalls erhält `fp` den Wert `NULL`. Konnte die Funktion gefunden werden, so wird sie mittels `fp` zur Ausführung gebracht (Zeile 20), ansonsten wird eine Fehlermeldung ausgegeben (Zeile 22). Durch den Aufruf von `ksymdone()` in Zeile 25 wird der durch `ksyminit()` in Zeile 09 für die Symboltabelle der Linker-Applikation reservierte Speicherplatz wieder freigegeben. In Zeile 26 wird das Ende des Blocks erreicht, in dem das `Module`-Objekt `A` gültig ist. An dieser Stelle wird der Terminierungscode der zu `A` gehörigen Objektmodule ausgeführt und die Module werden aus dem Speicher entfernt. Code-Beispiel A.2 liegt im Verzeichnis `demo/` als Datei `odlappA.C` (in leicht veränderter) Form vor. Das entsprechende Programm wird durch den Compiler-Aufruf

```
$ gcc -static -o odlappA odlappA.C -lodl
```

erzeugt (für Details s. `Makefile`). Durch die Option `-lodl` wird der dynamische Linker in Form der Datei `libodl.a` statisch an das zu erstellende Programm gebunden. Das Verzeichnis `demo/` enthält darüber hinaus den Quelltext der folgenden geteilt benutzbaren Objektdateien[2]:

- `O, A, B, C, D, E`
  Diese Module formen zusammen den in Abb. A.2 dargestellten Abhängigkeitsgraphen.

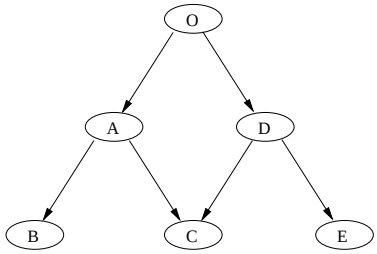


Abbildung A.2: Beispieldateien (1)

  Jedes dieser Objektmodule kann gelinkt werden. Dabei wird das Objektmodul zusammen mit allen benötigten Modulen geladen, mit `O` also würde der gesamte Graph aus Abb. A.2 geladen, mit `A` die Module `B` und `C`. Jedes dieser Module enthält ein globales Objekt der Klasse

[2]Bei der Darstellung der Dateinamen wird auf das Präfix `lib` und das Suffix `.C` bzw. `.so` verzichtet, kontextabhängig steht `X` also für `libX.C` oder `libX.so`

```
01:    #include <stdio.h>
02:    #include <odl.H>
03:
04:    typedef void(*Fp)(void);
05:
06:    int main(int argc, char *argv[]){
07:      Fp fp;
08:
09:      ksyminit(argv[0]);
10:
11:      Module A = "libA.so";
12:      if(A.geterrno() != DLE_NO_ERR){
13:        fprintf(stderr, "error linking libA.so: %s\n", A.geterrmsg());
14:        return(-1);
15:      }
16:
17:      A.dispinfo();
18:
19:      if(fp = (Fp) A.getsym("func(void)")){
20:        (fp)();
21:      } else{
22:        fprintf(stderr, "couldn't find func(void) in %s\n", A.getname());
23:      }
24:
25:      ksymdone();
26:    }
```

**Code–Beispiel A.2:** Beispiel für die Verwendung des dynamischen Linkers

`Trace` (s. `trace.H`). Durch Ausschriften von Konstruktor und Destruktor dieser Klasse kann die Ausführung von Initialisierungs- und Terminierungscode nachvollzogen werden. Weiterhin ist in jedem dieser Module eine initialisierte Integer-Variable `O_int`, `A_int` usw. und eine Funktion `O_func(void)`, `A_func(void)` usw. enthalten. Damit kann der Zugriff auf Variablen und der Aufruf von Funktionen ausprobiert werden. Die Funktionen rufen sich untereinander wie folgt auf:

- `O_func()` ruft `A_func()` und `D_func()`
- `A_func()` ruft `B_func()` und `C_func()`
- `D_func()` ruft `C_func()` und `E_func()`

Modul `C` ist innerhalb des durch `O` repräsentierten Abhängigkeitsgraphen ein Beispiel für eine mehrfach referenzierte Datei. Es enthält zusätzlich zum oben Gesagten eine nichtinitialisierte Variable `C_cmn` (Test der Belegung nichtinitialisierter Variablen) und eine Funktion `C_func(int)` (Test der Parameterübergabe, in Verbindung mit `C_func(void)` Test der Typsicherheit). Modul `E` beinhaltet zusätzlich zum oben Gesagten die Definition einer Klasse `Footware` und ein globales Objekt `shoe` dieser Klasse. Anhand dieses Objektes kann der Zugriff auf Methoden einer Klasse ausprobiert werden. Weiterhin enthält `E` eine Funktion `E_link(void)`, bei deren Aufruf das Modul `B` gelinkt und daraus die Funktion `B_func(void)` aufgerufen wird (Test des dynamischen Linkens aus dynamisch gelinkten Modulen heraus).

- `G1, G2`

  Diese Module dienen der Demonstration der globalen Symbolsuche, sie sind nicht in einem gemeinsamen Abhängigkeitsgraphen enthalten. Modul `G1` definiert eine Funktion `bicycle(void)`, die durch die Funktion `fish(void)` aus Modul `G2` aufgerufen wird. Bei der Relokation von `G2` wird die Symboldefinition für `bicycle(void)` nur dann gefunden, wenn `G1` zuvor zur globalen Symbolsuche freigegeben wurde.

- `S11, S12, S21, S22`

  Diese Module dienen der Demonstration der Verwendung von Gültigkeitsbereichen. Die Module `S11` und `S12` formen einen Abhängigkeitsgraphen mit `S11` als Wurzel, `S21` und `S22` bilden einen weiteren Abhängigkeitsgraphen mit `S21` als Wurzel. Die Module `S11` und `S21` definieren beide eine initialisierte Variable `i` und eine Funktion `foo(void)`, die eine Funktion `bar(void)` aufrufen. Die Module `S12` und `S22` definieren beide eine solche Funktion `bar(void)`, die ihrerseits auf eine Variable `i` zugreift. Durch entsprechende Ausschriften der Funktionen kann überprüft werden, ob Symbolreferenzen mit den Symboldefinitionen des jeweiligen Abhängigkeitsgraphen aufgelöst wurden.

- `VA, VB, VC, VD, VE, VF`

  Diese Module dienen der Demonstration der *relink*-Operation und der Versionsverwaltung. Die Module `VA`, ..., `VE` entsprechen im wesentlichen den Modulen `A`, ..., `E` (s. o.). Von dem durch `VA` repräsentierten Abhängigkeitsgraphen können die Versionen 1.1, 1.2, 1.3, 1.4 und 2.0 erstellt werden (s. `Makefile`). In der folgenden Aufzählung wird die (sukzessive) Verwendung der Versionen erläutert[3] . Der Ausgangspunkt der Betrachtungen ergibt sich durch das Laden der Module `VA` (Version 1.1) und `VD`. Nach der Erzeugung der nächsten Version werden die Module durch eine *relink*-Operation über `VA` aktualisiert. Durch den Aufruf der Funktionen `VA_func(void)` bzw. `VD_func(void)` kann die richtige Verwendung von Symboldefinitionen überprüft werden. Die Ausschriften der Funktionen enthalten Hinweise auf die jeweilige Version des Objektmoduls.

  – **Version 1.1**

    Version 1.1 des durch `VA` repräsentierten Abhängigkeitsgraphen stellt die Ausgangssituation der folgenden Schritte dar. (s. Abb. A.3).

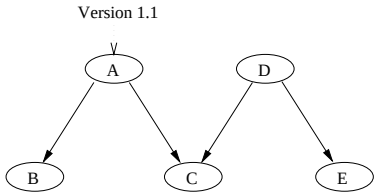


Abbildung A.3: Beispieldateien (2)

  – **Version 1.1 → Version 1.2**

    In Version 1.2 wurden gegenüber Version 1.1 die Werte globaler Variablen und die Ausschriften der Funktionen verändert. Die Struktur des Abhängigkeitsgraphen ist gleich geblieben (s. Abb. A.4).

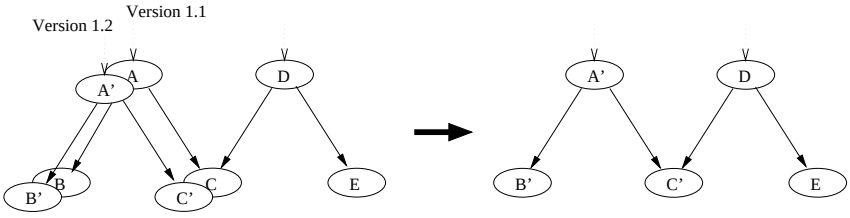


Abbildung A.4: Beispieldateien (3)

  – **Version 1.2 → Version 1.3**

    In Version 1.3 entfällt gegenüber Version 1.2 die Referenz von `VA` zu `VC` (s. Abb. A.5). Da `VC` aber weiterhin von `VD` benötigt wird, muß es als Bestandteil des durch `VD` repräsentierten Abhängigkeitsgraphen erhalten bleiben.

[3] In den Abbildungen wurde auf das führende „V“ im Modulnamen verzichtet, so steht `A` z. B. für `VA`.

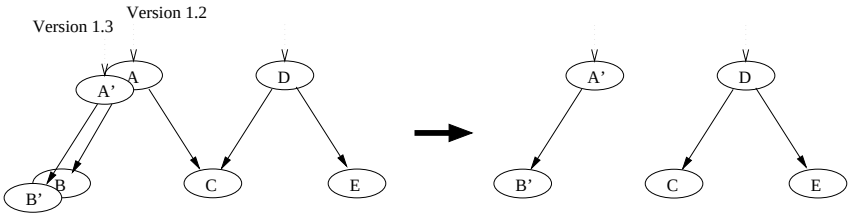


Abbildung A.5: Beispieldateien (4)

– **Version 1.3 → Version 1.4**

In Version 1.4 übernimmt `VA` gegenüber Version 1.3 wieder eine Referenz zu `VC`. Module `VC` referenziert in dieser Version ein weiteres Modul `VF` (s. Abb. A.6). Modul `VF` definiert eine Funktion `VF_func(void)`, die durch die Funktion `VC_func(void)` aus `VC` aufgerufen wird. Zu beachten ist, daß die neue Version von `VC` auch in dem durch `VD` repräsentierten Abhängigkeitsgraphen verwendet wird.

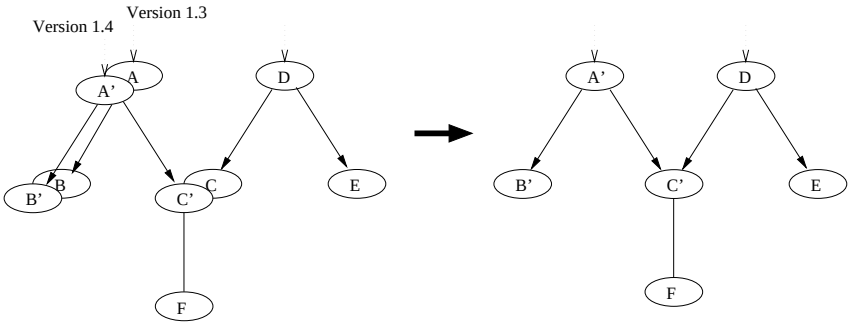


Abbildung A.6: Beispieldateien (5)

– **Version 1.4 → Version 2.0**

Die Module der Version 2.0 sind nicht mehr mit denen der Version 1.4 kompatibel (andere Major-Versionsnummer). Sie dürfen daher nur in dem durch `VA` repräsentierten Abhängigkeitsgraphen verwendet werden, während die Module `VC` und `VF` (Version 1.4) in dem durch `VD` repräsentierten Abhängigkeitsgraphen erhalten bleiben (s. Abb A.7). Zu beachten ist, daß nach dem Wechsel zur Version 2.0 zwei (inkompatible) Versionen von `VC` verwaltet werden.

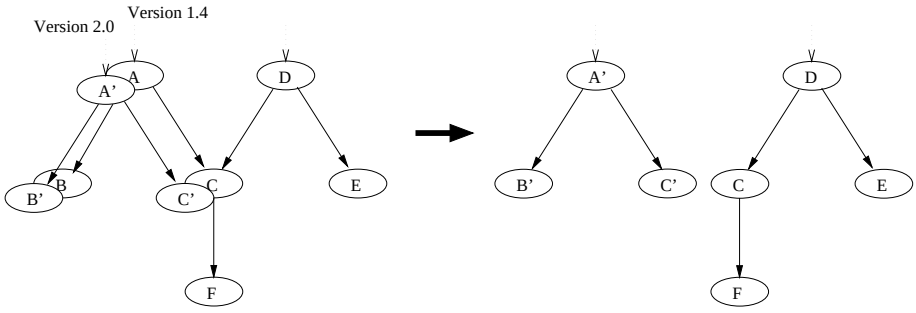


Abbildung A.7: Beispieldateien (6)

- `X, Y, Z`

Der durch diese Module geformte Abhängigkeitsgraph verkörpert eine sogenannte *implizite Referenz* (s. Abb A.8). In diesem Graphen könnte – ohne die Auflösbarkeit symbolischer Referenzen

einzuschränken – die Kante zwischen `X` und `Z` oder zwischen `Y` und `Z` entfernt werden. Dieses Beispiel dient zur Überprüfung des vom dynamischen Linker verwendeten BFT-Algorithmus.

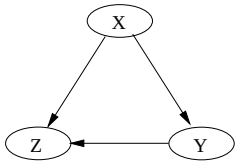


Abbildung A.8: Beispieldateien (7)

- `err*`

  Diese Module dienen zur Demonstration der Reaktion des dynamischen Linkers auf verschiedene Fehlerbedingungen. Ihr vollständiger Name bildet sich nach der entsprechenden Fehlernummer (s. Tabelle A.4, S. 94).

Im Verzeichnis `demo/` sind darüber hinaus die Quelltexte der folgenden Beispielprogramme enthalten[4]:

- `odlsh`

  Hinter diesem Programm verbirgt sich eine „Mini-Shell“, mit der die Funktionen des dynamischen Linkers interaktiv benutzt werden können (s. `help` am Kommando-Prompt).

- `odlappA`

  (entspricht Code-Beispiel A.2, s.o.)

- `odlappAD`

  Dieses Programm entspricht im wesentlichen `odlappA`. In einem eingenesteten Block wird jedoch zusätzlich das Modul `D` geladen und eine darin enthaltene Funktion aufgerufen.

- `odlappE`

  Durch dieses Programm wird das Modul `E` geladen und die darin enthaltene Funktion `E_link()` (s. o.) aufgerufen.

- `odlappG`

  Dieses Programm demonstriert die globale Symbolsuche anhand der Module `G1` und `G2` (s. o.).

- `odlappR`

  Durch dieses Programm wird der durch `A` repräsentierte Abhängigkeitsgraph und damit auch das Modul `C` geladen. Anschließend wird dem Nutzer Gelegenheit gegeben, das Modul `C` zu modifizieren. Danach wird mittels einer *link*-Operation ein Identifikator für `C` erworben. Mittels dieses Identifikators wird eine *relink*-Operation über `C` ausgeführt. Funktionsaufrufe vor und nach der *relink*-Operation machen die Veränderungen an `C` deutlich.

[4]Bei der Darstellung der Dateinamen wird auf das Suffix `.C` verzichtet, `odlX` steht kontextabhängig also für `odlX.C` oder das daraus erzeugte Programm

# Anhang B

# Initialisierungs- und Terminierungscode

## B.1 Initialisierungs- und Terminierungscode mit dem Sun-C++-Compiler

Der Sun-C++-Compiler setzt die Konstruktor- und Destruktoraufrufe globaler Objekte nicht direkt in die `.init`- und `.fini`-Sektionen, sondern erzeugt pro Objektmodul die folgenden Funktionen:

- `void __0CH_STCON_v(void)`[1]
  In dieser Funktion werden die Konstruktoraufrufe globaler Objekte des Moduls akkumuliert. Sie soll im folgenden als *Meta-Konstruktor* bezeichnet werden.

- `void __0DH_STDES_v(void)`[2]
  In dieser Funktion werden die Destruktoraufrufe globaler Objekte des Moduls akkumuliert. Sie soll im folgenden als *Meta-Destruktor* bezeichnet werden.

Sowohl Meta-Konstruktor als auch Meta-Destruktor sind lokal gebundene Funktionen. Dadurch kann ein eventueller Namenskonflikt beim Linken mehrerer Objektdateien mit globalen Objekten verhindert werden. Die Aufrufe von Meta-Konstruktor und -Destruktor werden in die `.init`- und `.fini`-Sektionen geschrieben. Durch die Pragmas `init` und `fini` können die Aufrufe weiterer Funktionen in die `.init`- bzw. `.fini`-Sektion gesetzt werden (s. Code-Beispiel B.1). Das Code-Beispiel B.2 zeigt mittels (Pseudo-) Assembler-Code den vom Compiler generierten Inhalt der `.init`- und `.fini`- Sektion. Beide Sektionen verfügen nach der Erzeugung durch den Compiler weder über einen Funktionsprolog noch über einen Funktionsepilog (`return`-Befehl), und können daher nicht als Funktion interpretiert werden. Erst bei der Erzeugung einer geteilt benutzbaren Objektdatei wird durch den Compiler-Treiber ein Aufruf des Link-Editors generiert, bei dem die Liste der Eingabedateien durch die Objektmodule `crti.o` und `crtn.o` geklammert wird:

```
$ ld -o itc.so crti.o itc.o ... crtn.o
```

Die Datei `crti.o` beinhaltet eine `.init`- und `.fini`-Sektion, die beide einen Funktionsprolog enthalten. Der Beginn beider Sektionen wird mit den Funktionssymbolen `_init` bzw. `_fini` markiert. Die `.init`- und die `.fini`-Sektion der Datei `crtn.o` enthalten jeweils einen Funktionsepilog. Durch den Link-Editor werden die `.init`- und `.fini`-Sektionen aller Eingabe-Objektdateien konkateniert. Dadurch werden diese Sektionen in der Ausgabedatei durch einen Funktionsprolog eröffnet und mit einen

[1] demangled: `__STATIC_CONSTRUCTOR(void)`
[2] demangled: `__STATIC_DESTRUCTOR(void)`

```
...
void initialize(){
  printf("trace: initialize()\n");
}

void terminate(){
  printf("trace: terminate()\n");
}

#pragma init (initialize)
#pragma fini (terminate)

complex example;
...
```

**Code–Beispiel B.1:** `itc.C`

```
...
.section ".init"
call    Meta-Konstruktor
call    initialize
...
.section ".fini"
call    Meta-Destruktor
call    terminate
...
```

**Code–Beispiel B.2:** `.init`- und `.fini`-Sektionen mit dem Sun-C++-Compiler

Funktionsepilog abgeschlossen. Der Inhalt dieser Sektionen kann somit als Funktion aufgerufen werden. Die relativen Adressen beider Funktionen werden anschließend vom Link-Editor in das Dynamic-Segment der Ausgabedatei eingetragen (Eintragstypen `DT_INIT` bzw. `DT_FINI`). Die Tabelle B.1 stellt die Konkatenation der Sektionen dar. Die Spalten der Tabelle geben die Eingabe-Objektdateien an, die Zeilen repräsentieren die (akkumulierten) Sektionen der Ausgabedatei.

| Sektion | crti.o | itc.o | crtn.o |
|---|---|---|---|
| .init | void _init(){<br>// Funktionsprolog | __OCH_STCON_v(); // Funktionsaufruf<br>initialize(); // Funktionsaufruf | return;}<br>// Funktionsepilog |
| .text | | static void __OCH_STCON_v(){ ... } // Funktionsdef.<br>static void __ODH_STDES_v(){ ... } // Funktionsdef.<br>void initialize(){ ... } // Funktionsdef.<br>void terminate(){ ... } // Funktionsdef. | |
| .fini | void _fini(){<br>// Funktionsprolog | __ODH_STDES_v(); // Funktionsaufruf<br>terminate(); // Funktionsaufruf | return;}<br>// Funktionsepilog |

Tabelle B.1: Initialisierungs- und Terminierungscode mit dem Sun-C++-Compiler

## B.2 Initialisierungs- und Terminierungscode mit dem GNU-C++-Compiler

Der GNU-C++-Compiler verfährt mit den Konstruktor- und Destruktoraufrufen globaler Objekte wie der Sun-C++-Compiler: Er generiert für jedes Objektmodul die folgenden Meta-Konstruktoren und -Destruktoren:

- `void _GLOBAL_.I.`*name*`()`
  Im konkreten Fall ist *name* durch den Namen des ersten im Objektmodul enthaltenen globalen Objekts zu ersetzen. Für das Code-Beispiel B.1 aus dem vorigen Abschn. heißt der Meta-Konstruktor also `_GLOBAL_.I.example`.
- `void _GLOBAL_.D.`*name*`()`
  Im konkreten Fall ist *name* durch den Namen des ersten im Objektmodul enthaltenen globalen Objekts zu ersetzen. Für das Code-Beispiel B.1 aus dem vorigen Abschn. heißt der Meta-Destruktor also `_GLOBAL_.D.example`.

Der Compiler setzt die Aufrufe dieser beiden Funktionen nicht direkt in die `.init`- bzw. `.fini`-Sektion. Stattdessen wird ein Zeiger auf den Meta-Konstruktor in die Sektion `.ctors` plaziert, während ein Zeiger auf den Meta-Destruktor in der Sektion `.dtors` abgelegt wird. Das Code-Beispiel B.3 zeigt mittels (Pseudo-) Assembler-Code den Inhalt dieser beiden Sektionen.

```
...
.section ".dtors"
.long    _GLOBAL_.D.example
...
.section ".ctors"
.long    _GLOBAL_.I.example
```

**Code–Beispiel B.3:** `.ctors`- und `.dtors`-Sektionen

Bei der Erzeugung einer geteilt benutzbaren Objektdatei wird durch den Compiler-Treiber ein Aufruf des Link-Editors generiert, bei dem die Liste der Eingabedateien durch die Objektmodule `crti.o` und `crtbegin.o` angeführt und durch die Objektmodule `crtend.o` und `crtn.o` abgeschlossen wird:

```
$ ld -o itc.so crti.o crtbegin.o itc.o ... crtend.o crtn.o
```

Dabei werden `crti.o` und `crtn.o` exakt wie beim Sun-C++-Compiler gebraucht. Die Datei `crtbegin.o` beinhaltet eine `.ctors`- und `.dtors`-Sektion, und markiert deren Beginn mit den Symbolen

- `__CTOR__LIST__` bzw.
- `__DTOR__LIST__`.

Die Datei `crtend.o` beinhaltet ebenfalls eine `.ctors`- und `.dtors`-Sektion, und markiert deren Ende mit den Symbolen

- `__CTOR_END__` bzw.
- `__DTOR_END__`.

Der Link-Editor konkateniert die `.ctors`-und `.dtors`-Sektionen aller Eingabe-Objektdateien. Dadurch stellen diese Sektionen in der Ausgabedatei Listen von Funktionszeigern dar, deren Anfang und Ende

mit den oben angegebenen Symbolen gekennzeichnet ist. In der Datei `crtbegin.o` wird darüber hinaus eine Funktion `__do_global_dtors_aux()` definiert. Durch diese Funktion wird die `__DTOR_LIST__` rückwärts durchmustert, wobei die darin referenzierten Funktionen ausgeführt werden. Die `.fini`-Sektion der Datei `crtbegin.o` enthält den Aufruf der Funktion `__do_global_dtors_aux()`. In der Datei `crtend.o` wird analog dazu eine Funktion `__do_global_ctors_aux()` definiert. Durch diese Funktion wird die `__CTOR_LIST__` durchmustert, wobei ebenfalls die darin referenzierten Funktionen ausgeführt werden. Die `.init`-Sektion der Datei `crtend.o` enthält den Aufruf der Funktion `__do_global_ctors_aux()`. Die Tabelle B.2 stellt die Konkatenation der Sektionen durch den Link-Editor dar. Die Spalten der Tabelle geben wiederum die Eingabe-Objektdateien an, die Zeilen repräsentieren die (akkumulierten) Sektionen der Ausgabedatei.

| Sektion | crti.o | crtbegin.o | itc.o | crtend.o | crtn.o |
|---|---|---|---|---|---|
| .init | void init(){<br>// Funktionsprolog | | | __do_global_ctors_aux();<br>// Funktionsaufruf | return;}<br>// Funktionsepilog |
| .ctors | | __CTORS_LIST__ | _GLOBAL_.I.example | __CTOR_END__ | |
| .dtors | | __DTORS_LIST__ | _GLOBAL_.D.example | __DTOR_END__ | |
| .text | | static void<br>__do_global_dtors_aux()<br>{ ... } // Funktionsdef. | void _GLOBAL_.I.example()<br>{ ... } // Funktionsdef.<br>void _GLOBAL_.D.example()<br>{ ... } // Funktionsdef. | static void<br>__do_global_ctors_aux()<br>{ ... } // Funktionsdef. | |
| .fini | void _fini{<br>// Funktionsprolog | __do_global_dtors_aux();<br>// Funktionsaufruf | | | return;}<br>Funktionsepilog |

Tabelle B.2: Initialisierungs- und Terminierungscode mit dem GNU-C++-Compiler

# Thesen

1. Das dynamische Linken ermöglicht die Adaption von Software-Systemen zur Laufzeit durch das dynamische Laden und Entfernen von Programmkomponenten.
2. Dynamisches Linken ist durch die Auflösung symbolischer Referenzen während des Programmstarts bzw. während der Programmausführung gekennzeichnet.
3. Objektdateien einer dynamisch gelinkten Anwendung stellen eine logische Einheit dar, die vom Laufzeit-Linker zu einer physischen Einheit (ausführbarer Programmcode) verbunden werden.
4. Das ELF-Objektdateiformat ist gut für das dynamische Linken geeignet.
5. Durch das dynamische Linken wird die geteilte Benutzung von Objektmodulen durch mehrere Anwendungsprogramme ermöglicht.
6. Der Mechanismus des positionsunabhängigen Codes erlaubt die wahlfreie Positionierung von geteilt benutzbaren Objektmodulen im Adreßraum eines Prozesses.
7. Typsicherheit kann auf die Übereinstimmung vom Symbolnamen zurückgeführt werden.
8. Symbolgranulares Linken ist auf der Basis existierender Objektdateiformate nicht möglich.
9. Abhängigkeitsgraphen stellen eine geeignete Datenstruktur zur Durchführung des dynamischen Linkens dar. Die entsprechenden Funktionen lassen sich als Operationen über Abhängigkeitsgraphen darstellen.
10. Klassenhierarchien können auf Objektmodulhierarchien (Abhängigkeitsgraphen) abgebildet werden.

# Selbständigkeitserklärung

Hiermit versichere ich, daß ich die vorliegende Diplomarbeit selbständig und ohne unzulässige fremde Hilfe verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Chemnitz, den 30. August 1996 Andreas Eulitz