Prozesserzeugung
Linker, Lader, Bibliotheken

Hans Jürgen Ohlbach

22. November 2017

Keywords: Compiler, Linker, Lader, Bibliotheken, Plugins

Empfohlene Vorkenntnisse: Prozesse, virtuelle Speicherverwaltung mit Paging

Inhaltsverzeichnis

1 Einführung **2**

2 Basistechniken **3**

2.1 Relocation 3

2.2 Code Sharing und Copy on Write 4

3 Compiler, Linker, Lader **5**

4 Bibliotheken **6**

4.1 Statische Bibliotheken 6

4.2 Dynamische Bibliotheken 7

5 Bibliotheken zwischen Prozessen sharen **8**

6 Plugins **9**

7 Programmiersprachen mischen **9**

1 Einführung

In einem vereinfachten Modell geht man davon aus, dass ein Programm compiliert wird, der komplette Maschinencode auf der Festplatte liegt, und dann als Prozess gestartet werden kann. Falls man virtuelle Speicherverwaltung mit Paging nutzt, dann kann jedes Programm davon ausgehen, dass der Adressbereich bei Adresse 0 beginnt. Durch den Mechanismus der Seitentabellen werden dann die virtuellen Adressen auf die physischen Adressen abgebildet.

Das ist aber leider ein zu vereinfachtes Modell. Was man wirklich möchte ist:

- größere Programme erstellen, indem einzelne Teile getrennt editiert und compiliert werden,
- einzeln compilierte Teile zu einem lauffähigen Programm zusammenbinden,
- lauffähige Programmfiles so klein wie möglich halten, d.h.
  - wiederverwendbare Programmteile als Bibliotheken organisieren
  - Bibliotheken erst zur Startzeit des Programms dazu laden,
- *Plugins* noch im laufenden Programm nachladen,
- Code, welcher von mehreren Prozessen benutzt wird, möglichst nur einmal im Speicher haben,
- Programmteile aus verschiedenen Programmiersprachen zusammenbinden.

Zeiten:
Die Verarbeitung eines Programms geschieht jetzt in verschiedenen Schritten. Wir unterscheiden daher vier verschiedene Zeiten:

Compilezeit: Der *Compiler* compiliert einzelne Programmteile, die aber dann noch nicht lauffähig sind. Sie werden in sog. *Objektfiles* abgespeichert.

Linkzeit: Der *Linker* verknüpft die einzelnen compilierten Programmteile und evtl. statische Bibliotheken zu einem lauffähigen Programm.

Ladezeit: Der *Lader* lädt ein lauffähiges Programm, bindet evtl. dynamische Bibliotheken hinzu, und erzeugt den Prozess.

Laufzeit: Scheduler und Dispatcher kontrollieren die Ausführung der Programme. Evtl. wird der Lader nochmal gebraucht, um zur Laufzeit Plugins hinzuzubinden.

2 Basistechniken

Zunächst sehen wir uns bestimmte Techniken an, die für das erreichen der obigen Ziele gebraucht werden.

2.1 Relocation

Wenn der Compiler einzelne Teiles eines Programms separat compiliert, dann muss er Vorkehrungen für das nachfolgende Zusammenbinden der Teile treffen:

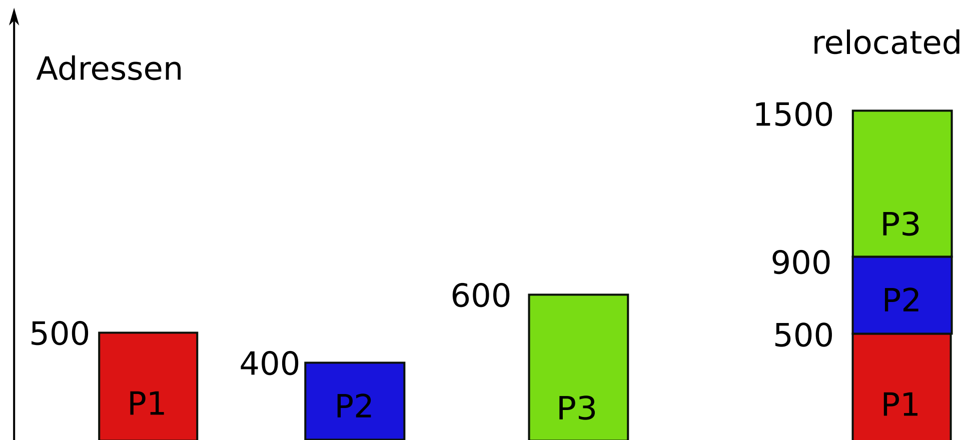
- Nur ein einziger Programmteil kann im zusammengebundenen Programm weiterhin bei Adresse 0 beginnen. In allen anderen Programmteilen müssen die Adressen um einen bestimmten Betrag verschoben werden.
- Ruft ein Programmteil P1 eine Funktion f in einem Programmteil P2 auf, dann kann der Compiler nicht wissen, wo diese Funktion im Maschinencode von P2 steckt. Er kann also keine konkrete Sprungadresse erzeugen. Daher muss diese Stelle markiert werden, um erst später die richtige Adresse dort einsetzen zu können.

Jeder Programmteil, der separat compiliert wird, beginnt weiterhin bei Adresse 0. Zusätzlich wird aber eine *Relocation Table* erzeugt. Hierin werden alle Stellen innerhalb des Objektfiles aufgelistet, an denen Adressen angepasst werden müssen. Zusätzlich wird gespeichert, an welchen Stellen welche Funktionen von anderen Programmteilen aufgerufen werden. Um dafür später die richtigen Adressen zu finden, muss man zusätzlich in jedem Programmteil eine Tabelle für die dort vorhandenen Funktionen mit ihren Adressen abspeichern.

Zwei Programmteile P1 und P2 kann man dann folgendermaßen zusammenzubinden:

- Die Adressierung von P1 ab Adresse 0 wird zunächst so belassen, wie sie ist.
- Für P2 gibt es jetzt eine Adressverschiebung, die sich aus der Länge von P1 ergibt.
- Zu allen Adressen von P2 muss diese Verschiebung hinzu addiert werden.
- In P1 müssen alle Aufrufe von Funktionen aus P2 durch die tatsächlichen Adressen ersetzt werden, und umgekehrt.

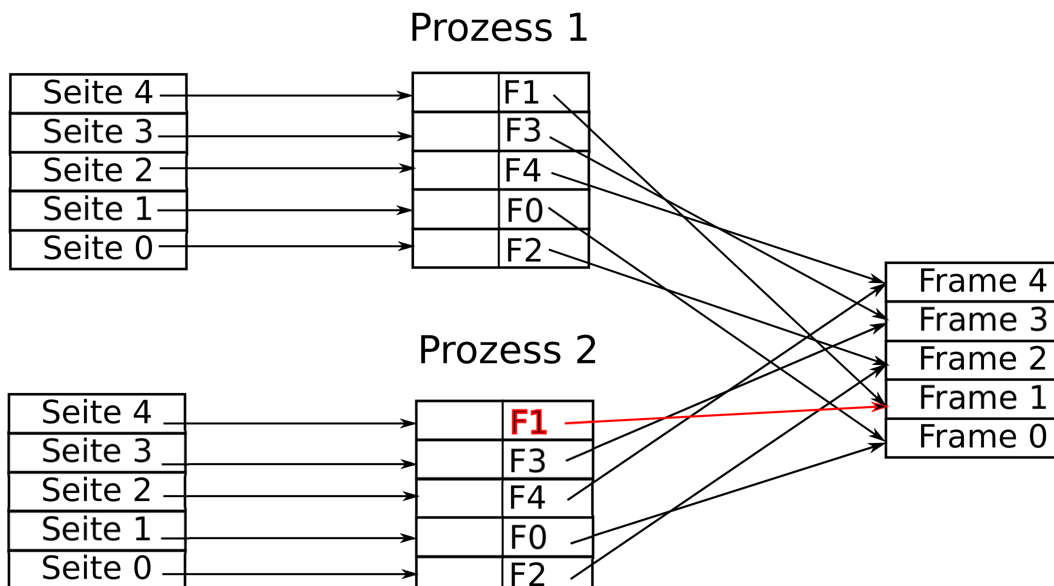
Den Vorgang bezeichnet man als *Relocation*. Er funktioniert natürlich auch, wenn mehr als zwei Programmteile zusammengebunden werden, wie in folgendem Bild:



2.2 Code Sharing und Copy on Write

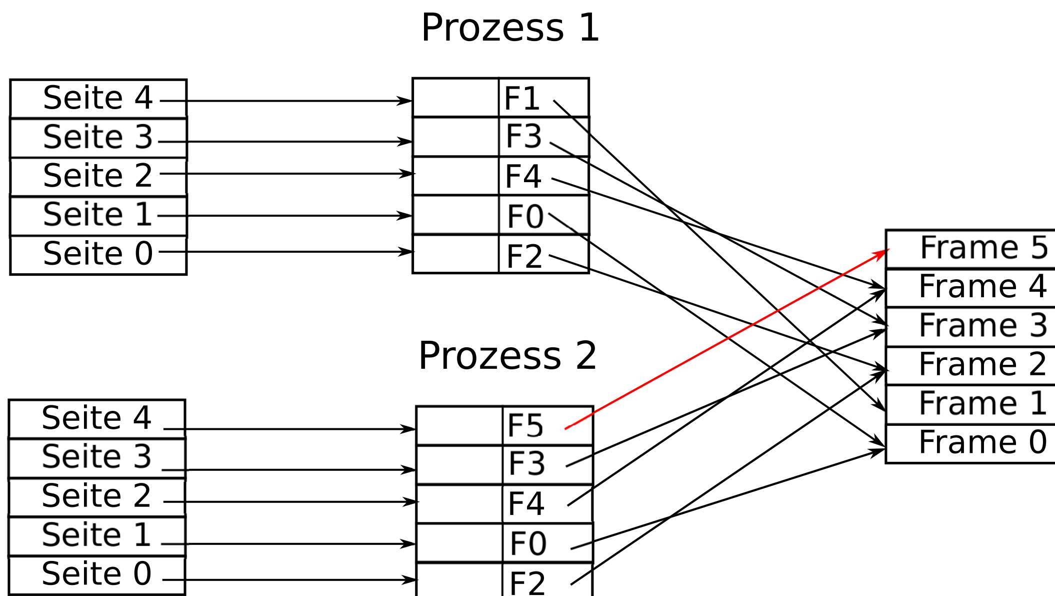
Mehrere der obigen Wünsche erfordern, dass unterschiedliche Prozesse Teile des Codes gemeinsam haben dürfen (Code Sharing). Das trifft insbesondere auf das „forken“ von Prozessen zu. *Copy on Write* hilft, dass die gemeinsamen Teile im Speicher nicht verdoppelt werden müssen, aber trotzdem jeder Prozess individuell Teile überschreiben darf.

Das funktioniert folgendermaßen: Angenommen, der gemeinsam genutzte Teil von Prozess 1 und Prozess 2 liegt in den Frames 0-4. Beide Prozesse haben eine eigene Seitentabelle, so dass die virtuellen Adressen gleich oder unterschiedlich sein können. Z.B. könnte virtuelle Seite 4 von Prozess 1 auf Frame 1 zeigen, und die virtuelle Seite 4 von Prozess 2 auch auf Frame 1 zeigen.



Angenommen, Prozess 2 will jetzt Daten in Seite 4, d.h. Frame 1 verändern. Damit Prozess 1 nichts davon mitbekommt, wird in diesem Moment Frame 1 kopiert, auf, z.B. Frame 5. Der Eintrag Frame 1 in der Seitentabelle von Prozess 2 wird dann auf Frame 5 geändert. Jetzt haben beide Prozesse eigene

Kopien des Frames und können dort Daten nach Belieben ändern. Die anderen 4 Frames betrifft das nicht. Sie können weiter „geshared“ werden.



Copy on Write kommt insbesondere beim „forken“ von Prozessen zum Einsatz. Das „forken“ eines Prozesses bedeutet dann nicht, dass der komplette Speicherbereich des Prozesses kopiert werden muss. Mit der virtuellen Speicherverwaltung genügt es, zunächst den Prozesskontrollblock und die Seitentabelle zu kopieren. Beide, Vater- und Kindprozess haben dann zunächst identische Seitentabellen und benutzen damit exakt die gleichen Bereiche im Arbeitsspeicher und Swapspace.

Copy on Write bedeutet in diesem Fall, dass, sobald einer der beiden Prozesse Daten in einem der von beiden benutzten Frames ändert, dieses Frame kopiert wird, und der Eintrag in seiner Seitentabelle dann auf das neue Frame zeigt. Das kopierte Frame gehört dann nur dem einen Prozess, und kann von ihm verändert werden, ohne dass der andere Prozess das mitbekommt. Natürlich wird dann auch eine entsprechende Kopie des Frames im Swapspace der Festplatte angelegt.

Damit müssen nur die wirklich unterschiedlichen Datenbereiche kopiert werden. Insbesondere der Bereich des Codes wird i.A. dann nicht kopiert.

3 Compiler, Linker, Lader

Compiler (und Assembler) übersetzen den Quellcode von separaten Programmteilen und erzeugen einen Objektfile. Die Adressierung beginnt immer bei 0. Zusätzlich wird aber eine Relocation Table erzeugt, in der all die Stellen aufgelistet sind, an denen Adressen geändert werden müssen.

Linker relocaten alle Programmteile und binden sie zusammen zu einem neuen Objektfile.

Lader laden einen Objektfile und formen daraus einen Prozess.

Sowohl beim Linken als auch beim Laden müssen Vorkehrungen getroffen werden, um Bibliotheken mit hinzuzubinden.

4 Bibliotheken

Häufig benutzte Programmteile werden meist in *Bibliotheken* zusammengefasst. Diese werden compiliert und in speziellen Objektfiles gespeichert. Der Linker könnte beim Linken aller Programmteile auch die Bibliotheken mit hinzufügen, so dass ein kompletter Objektfile entstünde, der für sich alleine lauffähig wäre. Damit würde eine Bibliothek aber unnötig vervielfacht. Eine Kopie einer Bibliothek würde eigentlich für alle Programme in einem Computer ausreichen. Trotzdem könnte es sinnvoll sein, in einen Objektfile, welchen man an Kunden ausliefert, alle Bibliotheken mit hinzuzubinden.

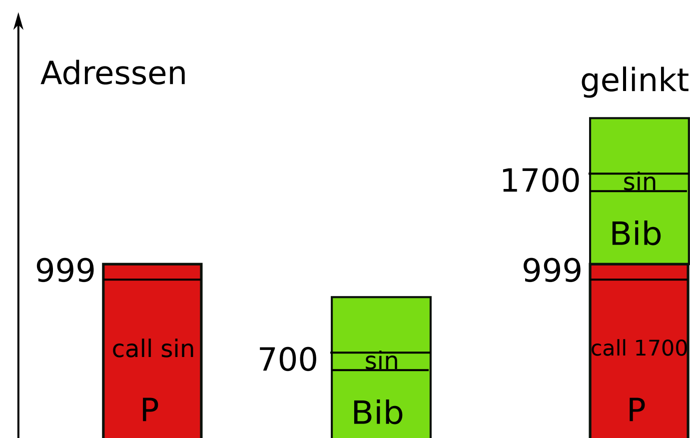
Man unterschiedet *statische Bibliotheken* und *dynamische Bibliotheken*.

4.1 Statische Bibliotheken

Bei einer statischen Bibliothek wird zur Linkzeit ein fester Adressraum für die Bibliothek festgelegt. Die Bibliothek wird aber *nicht* mit in den Objektfile eingebunden.

Als Beispiel nehmen wir eine mathematische Bibliothek mit der Sinusfunktion. Angenommen, die Sinusfunktion liegt an Adresse 700 in der Bibliothek. Angenommen, ein Programm P, welches die Sinusfunktion aufruft, benutzt Adressen von 0 bis 999. Das bedeutet, der Adressraum der Bibliothek muss um 1000 verschoben werden, so dass die Sinus-Funktion an Adresse 1700 zu liegen kommt. Jetzt kann der Linker überall, wo in P Aufrufe der Sinusfunktion sind, einen Sprung zu Adresse 1700 einbauen.

Der Linker erzeugt jetzt einen Objektfile für das Programm P, *ohne die Bibliothek selbst*, wo aber die Aufrufe der Sinusfunktion über die feste Adresse 1700 eingebaut sind. Dieser Objektfile kann jetzt an Kunden ausgeliefert werden. Diese müssen aber auch die Bibliothek selbst besitzen.



Der Lader hat jetzt, neben allen anderen, folgende Aufgaben:

- Er muss überprüfen, ob die Bibliothek schon im Arbeitsspeicher ist. Falls nicht, muss er sie in den Arbeitsspeicher laden.
- Er muss die Seitentabelle für den neuen Prozess um die Seiten für die Bibliothek erweitern, so dass die vom Linker eingebauten Sprungadressen in die Bibliothek auf die richtigen Framenummern verweisen.

In Windows sind Files mit der Endung .lib statische Bibliotheken. In Linux und Unix sind es Files mit der Endung .a.

Ein Problem: Angenommen, die statische mathematische Bibliothek hat die Version 1. Irgendwann kommt eine Version 2 auf den Markt, die neue Funktionen enthält. Es könnte daher sein, dass die Sinusfunktion jetzt nicht mehr an der Adresse 700 liegt, sondern an der Adresse 800. Benutzt ein Kunde jetzt den alten Objektfile des Programms P zusammen mit der neuen Version der Bibliothek, kann es nicht mehr funktionieren.

Will man also statische Bibliotheken unterstützen, dann müssen mit einer neuen Version der Bibliothek alle Programme neu gelinkt werden und neue Objektfiles ausgeliefert werden. Um dieses Problem zu lösen, hat man *dynamische Bibliotheken* eingeführt.

4.2 Dynamische Bibliotheken

Für diese Bibliotheken werden die Adressen nicht zur Linkzeit, sondern erst zur Ladezeit festgelegt. Ruft also ein Programm P die Sinusfunktion auf, dann enthält der gelinkte Objektfile eine Relocation Table, wo alle Positionen, an denen die Sinusfunktion aufgerufen wird aufgelistet sind. Es wird dann aber keine konkrete Adresse der Sinusfunktion gespeichert, sondern nur deren Name.

Jetzt muss der Lader zur Ladezeit in der aktuellen Version der Bibliothek die Position der Sinusfunktion bestimmen (engl. symbol resolution) und dann die relocatede Adresse in den Code des Programms P einführen. Das macht das Laden eines Programms aufwendiger. Aber man ist unabhängig von der Versionierung der Bibliothek. Es kann nur dann noch Probleme geben, wenn eine Funktion völlig verschwindet. Um das zu vermeiden, lässt man in neueren Versionen auch alte Funktionen noch drin, markiert sie in der Dokumentation aber als *deprecated*.

In Windows sind Files mit der Endung .dll dynamische Bibliotheken (Dynamically Linked Libraries). In Linux und Unix sind es Files mit der Endung .so (Shared Object).

Java:
In Java unterschiedet man nicht zwischen Programmfiles, statischen und dynamischen Bibliotheken. Alle compilierten Files sind .class Files. Sie wirken wie dynamische Bibliotheken. Allerdings wird die Symbol Resolution, d.h. symbolische Namen durch konkrete Adressen zu ersetzen, erst zur Laufzeit bei der ersten Verwendung einer Methode gemacht (erst dann werden die .class Files überhaupt erst

geladen). Methoden, die zwar in einem .class File drin sind, aber zur Laufzeit nicht gebraucht werden, erzeugen damit auch keinen Aufwand.

5 Bibliotheken zwischen Prozessen sharen

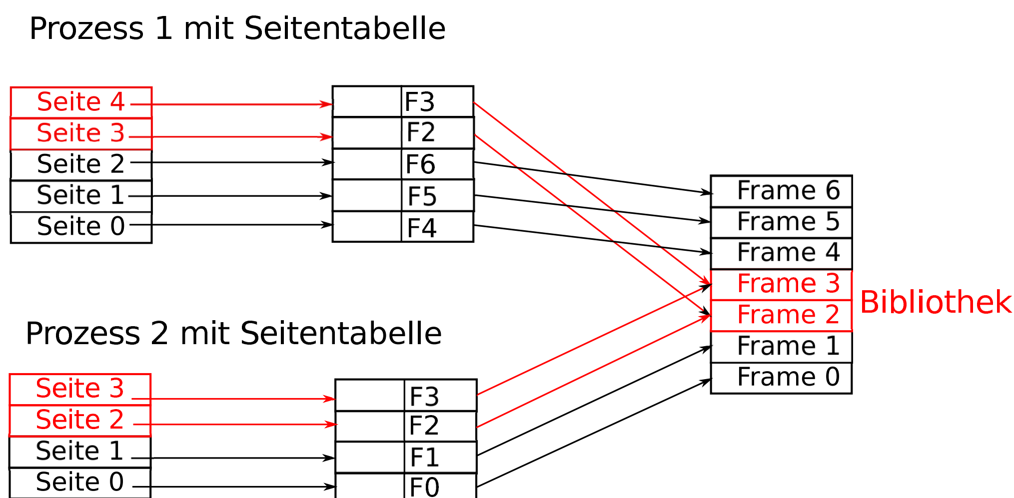
Es gibt viele Bibliotheken, die von vielen Programmen gebraucht werden. Wenn also 5 Programme gleichzeitig laufen und die gleiche Bibliothek verwenden, dann wäre es natürlich Platzverschwendung, wenn man die Bibliothek fünfmal kopieren müsste.

Jetzt hilft virtuelle Speicherverwaltung mit Paging, sowie Copy on Write, um *eine* Kopie der Bibliothek im Arbeitsspeicher von *mehreren* Prozessen verwendbar zu machen.

Dazu lädt sie das Betriebssystem beim ersten mal, wo die Bibliothek benötigt wird. Für jeden Prozess, der die Bibliothek braucht, wird eine entsprechende Erweiterung der Speichertabelle gemacht, wo die virtuellen Adressen, die die Bibliothek im Prozess einnimmt, auf die tatsächliche Framenummer der Bibliothek im Arbeitsspeicher abgebildet wird. Jetzt kann jeder Prozess seine eigenen virtuellen Adressen für die Bibliothek benutzen. Tatsächlich muss sie aber nur einmal im Speicher sein.

Das wird in der folgenden Graphik illustriert. Die Bibliothek liegt zunächst in den Frames 2 und 3. Es gibt zwei Prozesse, die diese verwenden. Bei Prozess 1 liegt der virtuelle Adressraum der Bibliothek in Seite 3 und 4, und bei Prozess 2 in Seite 2 und 3. Über ihre beiden Seitentabellen werden die Seiten aber beide auf die Frames 2 und 3 abgebildet.

Die Seiten 3 und 4 in Prozess 1 bzw. 2 und 3 in Prozess 2 existieren dabei nicht wirklich. Nur die Einträge in den Seitentabellen existieren wirklich. Bibliotheksaufrufe in den anderen Seiten (0,1,2 in Prozess 1 und 0,1 in Prozess 2) nutzen Adressen, deren Seitennummern in diesen nicht existierenden Seiten liegen. Diese Adressen werden beim Aufruf der Bibliotheksfunktionen über die Seitentabellen in die richtigen physikalischen Adressen übersetzt.



Durch Copy on Write wird darüber hinaus erreicht, dass jeder Prozess sogar Daten in der Bibliothek verändern darf, ohne dass das die anderen Prozesse stört.

6 Plugins

Plugins sind dynamische Bibliotheken, die erst zur Laufzeit hinzugeladen werden. Da das Hauptprogramm weder zur Compilezeit, noch zur Link- oder Ladezeit wissen kann, welches Plugin mit welcher Funktionalität hinzugeladen wird, müssen die Programme, die Plugins verwenden wollen, speziell dafür programmiert sein. Man braucht ein Protokoll, das festlegt, wie man mit den Plugins kommuniziert, und einen Plugin Manager, der Plugins zur Laufzeit laden und einbinden kann.

7 Programmiersprachen mischen

Sobald Quellprogramme in Maschinencode übersetzt sind, sollte es eigentlich egal sein, in welcher Programmiersprache der Quellcode geschrieben wurde. Die Maschinensprache ist ja immer die gleiche. Das Problem entsteht jedoch, sobald Funktionen aus jeweils anderen Sprachen aufgerufen werden müssen. Das macht schon dem Compiler Probleme. Wenn eine Funktion f aus Programmiersprache P1 von einem Programm aus Programmiersprache P2 aufgerufen werden soll, dann muss der Compiler von P2 wenigstens wissen, wieviele Argumente f hat, und welche Typen die Argumente haben. Innerhalb einer Sprache gibt es Konventionen, die der Compiler nutzen kann, z.B. die .h-Files in C und C++ mit den Spezifikationen der Funktionen.

Allein aus diesen Gründen ist es nicht einfach, Programmiersprachen zu mischen. Es gibt für einzelne Sprachen spezielle Mechanismen, die das unterstützen. In Java z.B. ist es das Java Native Interface. Damit kann man z.B. C-Programme aus Java aufrufen, und auch Java Methoden aus C.

Einen Mechanismus, der alle Kombinationen von Sprachen automatisch anwendbar macht, gibt es aber nicht.

Stichwortverzeichnis

Bibliothek, dynamisch, 7
Bibliothek, statisch, 6
Bibliotheken, 6

Code Sharing, 4
Compiler, 2, 5
Compilezeit, 2
Copy on Write, 4

Lader, 2, 5
Ladezeit, 2
Laufzeit:, 2
Linker, 2, 5
Linkzeit, 2

Plugins, 9

Relocation, 3
Relocation Table, 3