Verwendung und Implementierung von Bibliotheken

Lars Oergel

11. Februar 1999

Inhaltsverzeichnis

1 Motivation

Beim Implementieren größerer Systeme kommt es oftmals vor, daß man bestimmte Programmfragmente (sog. *Routinen*) an mehreren Stellen im Programm benutzt. Bestimmte Standardroutinen (z.B. für die Bildschirmausgabe) treten sogar in so gut wie jedem Programm auf. Um dabei das Rad nicht jedesmal neu erfinden zu müssen, kann man solche Routinen in Form von sog. Bibliotheken implementieren, d.h. in einer separaten Datei außerhalb des eigentlichen Programmcodes zusammenfassen. Alle Programme, die dann Funktionen aus einer solchen Bibliothek benutzen wollen, können beim Linken einfach mit[1] dieser Bibliothek gebunden werden.

Es gibt nun zwei verschiedene Ansätze zur Realisierung dieser Bibliotheken, die ich im folgenden etwas genauer[2], zunächst theoretisch, dann aber auch an einem Beispiel betrachten will, nämlich *statische* und *dynamische* Bibliotheken.

Unabhängig davon, welcher der beiden Ansätze zum Einsatz kommt, muß man natürlich die entsprechenden Funktionen innerhalb seines Programmes deklarieren[3]. Dieses geschieht i.d.R. durch ein `include`–Statement innerhalb des eigenen Programmtextes, das eine sog. *Headerdatei* einbindet, in der sämtliche Funktionen, die die Bibliothek zur Verfügung stellt, deklariert sind. Diese Headerdatei erhält man vom Programmierer der Bibliothek, d.h. wenn man selber eine Bibliothek schreiben will, muß man auch selbst in dieser Headerdatei die Schnittstelle der Bibliothek spezifizieren.

2 Statische Bibliotheken

Eine statische Bibliothek kann man erzeugen, indem man einfach mit Hilfe des `ar`–Befehls[4] alle Objektdateien, in denen Funktionen implementiert sind, die die Bibliothek zur Verfügung stehen soll, in einer Datei (einem sog. *Archiv*) zusammenfaßt. Zusätzlich legt der `ar`–Befehl eine Symboltabelle im Archiv ab, der der Linker entnehmen kann, in welcher Objektdatei welche Funktionen implementiert sind.

Beim Binden eines Programmes, das Funktionen der Bibliothek benutzen soll, wird dann mittels der Option `-l`*libname* festgelegt, in welcher Bibliothek die benutzten Routinen gefunden werden können. Per Konvention ist festgelegt worden, daß die entsprechende Datei auf den Namen `lib`*libname*`.a` horcht. Gesucht wird diese Datei in einem Standardsuchpfad für Bibliotheken[5], der durch die `-L`*path*–Option erweitert werden kann. Der Linker stellt dabei sicher, daß nur genau die Objektdateien aus der Bibliothek kopiert und zum Binary dazugebunden werden, die Funktionen implementieren, die das Programm auch wirklich benutzt.

Nachteil bei der Verwendung statischer Bibliotheken ist allerdings, daß der (per Definition ja immer gleiche) Programmcode der Bibliothek in jedes neu erzeugte Executable *kopiert* wird, was natürlich eine enorme Platzverschwendung sowohl auf der Festplatte als auch im Hauptspeicher darstellt. Um dieses Übel aus der Welt zu schaffen, hat man sich das Konzept der *dynamischen* Bibliotheken ausgedacht.

[1] Man sagt auch *gegen* eine Bibliothek binden.
[2] Wir betrachten hier „natürlich“ ein Linux System. :-)
[3] Der Compiler muß die sog. Signatur kennen, d.h. wie die Funktion heißt, mit welchen Parametern sie aufgerufen werden muß und was sie für einen Ergebnistyp zurückliefert.
[4] siehe auch `man ar`
[5] Dieser ist im Linker fest eingebaut und beinhaltet z.B. `/lib`, `/usr/lib` und `/usr/local/lib`.

3 Dynamische Bibliotheken

Bei dynamischen Bibliotheken wird der Code, der die Funktionen der Bibliothek implementiert, systemweit nur in einer einzigen Datei gespeichert. Programme, die dynamisch gegen diese Bibliothek gebunden sind, enthalten nur einen Verweis auf die benutzte Bibliothek, nicht aber den Bibliotheks–Programmcode selber. Bei jedem Aufruf eines solchen Programmes muß zunächst die entsprechende Bibliothek nachgeladen werden, wenn sie sich noch nicht im Hauptspeicher befindet, bevor das eigentliche Programm gestartet werden kann.

3.1 Vor- und Nachteile

Dynamische Bibliotheken bieten eine Menge Vorteile gegenüber statischen Bibliotheken:

- Der Programmcode der Bibliothek liegt nur ein einziges Mal auf der Festplatte (Platzersparnis).
- Er liegt auch nur einmal im Hauptspeicher, auch wenn er von mehreren Prozessen gleichzeitig benutzt wird (Zeitersparnis, da der Code nur einmal von der Festplatte geladen werden muß und Platzersparnis).
- Außerdem kann die Bibliothek relativ einfach upgedatet werden, z.B. bei Fehlerkorrekturen. Da alle Programme auf den Code in einer einzigen Datei zugreifen, braucht man nur diese Datei zu erneuern, und fortan wird der verbesserte Code von jedem Programm sofort benutzt. Bei Verwendung statischer Bibliotheken müßten alle Programme neu übersetzt werden, um den verbesserten Code benutzen zu können.

Nachteile ergeben sich durch eine etwas höhere Komplexität bei der Verwendung von dynamischen Bibliotheken, zudem wird ein neuer zentraler Schwachpunkt geschaffen: Da annähernd sämtliche Systemprogramme eines Linux–Rechners (selbst der `ls`–Befehl) zumindest gegen die C–Bibliothek („libc“ genannt) dynamisch gebunden sind, führt ein Versagen des Mechanismus' zum dynamischen Binden oder ein fehlgeschlagenes Update einer wichtigen Bibliothek dazu, daß so gut wie kein Befehl mehr ausgeführt werden kann!

3.2 Schnittstellenkompatibilität und der SO–Name

Eine wichtige Rolle bei dynamischen Bibliotheken spielt die Schnittstellenkompatibilität zwischen unterschiedlichen Versionen ein und derselben Bibliothek.

Wenn sich nämlich die Schnittstelle, die ja im Headerfile der jeweiligen Bibliothek deklariert ist, ändert, kann es sein, daß Programme, die noch mit der alten Version entwickelt und gebunden wurden, nach dem Update der Bibliothek nicht mehr funktionieren, da sich die Bibliotheksfunktionen nicht mehr so verhalten, wie sich die Anwendung das denkt oder im Zweifelsfall gar nicht mehr existieren.

Man gibt daher einer Bibliothek einen sog. „SO–Namen" („shared object"), der neben dem eigentlichen Namen der Bibliothek auch noch die *Hauptversionsnummer* enthält. Ändert nun der Programmierer der Bibliothek die Schnittstelle, sollte er auch diese Hauptversionsnummer um eins erhöhen, damit erkennbar ist, daß die neue Version inkompatibel zur alten ist. Außerdem können dadurch Bibliotheken mit unterschiedlichen Hauptversionsnummern ohne Probleme auf einem System koexistieren: Durch den anderen so–Namen sind sie leicht zu unterscheiden.

Der Linker vermerkt beim Binden eines Programmes den so–Namen und damit auch die Hauptversionsnummer im Executable, so daß jederzeit festgestellt werden kann, mit welcher Version das Programm erzeugt wurde.

Zusätzlich gibt man einer Bibliotheksversion noch eine *Nebenversionsnummer* und optional eine *Releasenummer*. Beispielsweise sollte also ein Programm, das mit der libc–5.2.10 übersetzt wurde, auch noch problemlos mit der libc–5.4.33 zusammenarbeiten können. Im `/lib`–Verzeichnis vieler Linux–Distributionen findet sich neben der libc–5.x.y meist auch noch die etwas ältere und inkompatible libc–4.x.y, die zur Verwendung kommt, wenn ein älteres Binary aufgerufen wird, das noch mit dieser Version der libc gebunden wurde.

3.3 Erzeugen und Benutzen

Um eine dynamische Bibliothek zu erzeugen, sind zwei Schritte notwendig:

1. Zunächst muß man die einzelnen Objektdateien unter Verwendung der Option `-fPIC`[6] übersetzen.
2. Anschließend kann man sie mit Hilfe spezieller Optionen und der Angabe des so–Namens (über den `gcc` als Frontend zum Linker `ld`) zu einer dynamischen Bibliothek zusammenfassen.

   Konkret: Um z.B. die Dateien `libhello1.so` und `libhello2.so` zu einer dynamischen Bibliothek mit dem so–Namen *libhello.so.0* zu linken, die in der Datei `libhello.so.0.0` gespeichert werden soll, ist folgender Befehl zu geben:

   `gcc -shared -Wl,-soname,libhello.so.0 libhello1.so libhello2.so -o libhello.so.0.0`.

Um dann diese Bibliothek benutzen zu können, muß man beim Binden wiederum nur `-l`*libname* angeben. Die Namenskonvention für dynamische Bibliotheken sieht dann `lib`*libname*`.so` als Dateiname der Bibliothek vor. Dieses ist dann ein Softlink entweder auf die eigentliche Datei mit kompletter Versionsangabe[7] oder auf einen weiteren Softlink, in dem nur die Hauptversionsnummer spezifiziert ist[8]. Auch hier wird wieder innerhalb des Standardsuchpfades und evtl. weiteren mit `-L`*path* angegeben Pfaden nach der Datei gesucht.

Es wird jedoch im Endeffekt nur der so–Name der Bibliothek aus der gefundenen Datei im Executable abgelegt. Für das Auffinden der Bibliothek beim Starten des Programmes ist ein anderer Mechanismus zuständig, der im nächsten Absatz beschrieben wird. Findet der Linker eine statische und eine dynamische Version, d.h. `.a`- und `.so`–Files der gesuchten Bibliothek, wird defaultmäßig dynamisch gebunden, es sei denn, man weist mit `-static` vor der `-l`*libname*–Angabe explizit das statische Binden an.

[6] PIC: „Position independent Code"
[7] z.B. `/lib/libc.so -> /lib/libc.so.5.4.33`
[8] z.B. `/lib/libc.so -> /lib/libc.so.5` und `/lib/libc.so.5 -> /lib/libc.so.5.4.33`

3.4 Der dynamische Lader ld–linux

Da der Umgang mit Programmen sehr mühselig wäre, wenn in jedem dynamisch gebundenen Executable der absolute Pfad der benötigten Bibliotheken abgelegt wäre, gibt es einen Mechanismus, der die zentrale Verwaltung des dynamischen Bindens und der abzusuchenden Pfade ermöglicht: Der dynamische Lader `ld-linux`[9].

Jedes dynamisch gebundene Programm kennt[10] den distributionsübergreifend standardisierten Pfad zum dynamischen Lader, `/lib/ld-linux.so.x` und des weiteren nur noch die so–Namen der von ihm benötigten Bibliotheken.

Beim Aufruf eines Programmes wird nun immer zuerst der dynamische Lader aufgerufen und ihm mitgeteilt, welche dynamischen Bibliotheken das Programm benötigt. Er überprüft, ob sich diese Bibliothek vielleicht schon im Hauptspeicher befindet und sucht und lädt gegebenenfalls die Bibliothek nach, bevor er die Kontrolle an das eigentliche Programm zurückgibt, das sich dann seinen Aufgaben widmen kann. Schlägt dieser Vorgang fehl, weil z.B. der Lader defekt ist, gar gelöscht wurde oder einfach die benötigte Bibliothek nicht finden oder nicht laden kann, bekommt man außer der lapidaren Meldung „`can't load library '`*libname*`.so.`*x*`'`" nichts zu sehen. Da das Programm gar nicht komplett geladen werden konnte (der Bibliothekscode fehlt ja), kann auch nicht eine einzige Zeile Programmcode abgearbeitet werden.

In der Datei `/etc/ld.so.conf` kann der Systemverwalter konfigurieren[11], in welchen Verzeichnissen `ld-linux` nach dynamischen Bibliotheken suchen soll. Möchte ein normaler Benutzer Bibliotheken außerhalb dieser Pfade benutzen können, kann er dem Lader weitere Pfade in der Umgebungsvariablen `LD_LIBRARY_PATH` mitteilen.

Damit `ld-linux` nicht bei jedem einzelnen Programmstart mehrere, u.U. recht gut gefüllte Verzeichnisse auf der Suche nach einer dynamischen Bibliothek abklappern muß, unterhält er sich einen Cache, in dem sämtliche Bibliotheken, die sich in seinem Suchpfad befinden, verzeichnet sind. Diesen Cache, der übrigens in der Datei `/etc/ld.so.cache` gespeichert wird, kann der Systemverwalter mit dem Befehl `ldconfig` nach einem Update oder einer Suchpfad–Änderung aktualisieren[12].

Aber auch der normale Systembenutzer kann sich vermittels `/sbin/ldconfig -v` einen einfachen Überblick über sämtliche auf dem System verfügbaren Bibliotheken verschaffen; der Cache wird dabei natürlich mangels geeigneter Rechte nicht verändert.

Außerdem legt `ldconfig` die benötigten Softlinks an bzw. hängt sie bei einem Update einer Bibliothek auf die jeweils neueste Version um. Dabei ist zu beachten, daß `ldconfig` nur die Softlinks anlegt bzw. ändert, die der `ld-linux` benötigt; d.h. er legt jeweils einen Softlink an, der vom so–Namen der Bibliothek auf die momentan aktuelle verfügbare Datei verweist. Zum Übersetzen und Binden eigener Programme ist jedoch i.d.R. ein weiterer Softlink nötig, der spezifiziert, welche Hauptversion einer Bibliothek verwendet werden soll, der Linker sucht ja z.B. nach `libc.so` und nicht nach `libc.so.5`.

[9] Strenggenommen ist `ld-linux` nur für Executables zuständig, die nach dem „ELF"–Binärformat gebunden worden sind. Programme, die noch nach dem alten „a.out"–Schema erzeugt wurden, werden vom `/lib/ld.so` bedient.
[10] wie leicht mit Hilfe des Befehls `string` *executable* festzustellen
[11] ein Verzeichnis pro Zeile
[12] Manche Distributionen machen das auch bei jedem Systemstart.

3.5 Zusammenfassung

Hier nochmal eine Zusammenfassung der Vorgänge beim dynamischen Binden, vom Kompiliervorgang bis zum Ausführen des Binaries:

1. Der Linker wird nach dem Übersetzen u.a. mit der Option `-lc` aufgerufen, d.h. er soll eine Datei `libc.so` suchen und das Executable dynamisch gegen die darin enthaltene Bibliothek binden.
2. Der Linker findet `libc.so` als einen „manuell“[13] anzulegenden Link, der auf `libc.so.5` verweist.
3. `libc.so.5` ist ein weiterer Softlink, der auf die Datei `libc.so.5.4.33` zeigt. Endlich ist der Linker bei der konkreten Bibliothek angekommen.
4. Der Linker ermittelt aus der Datei `libc.so.5.4.33` den so–Namen der darin befindlichen Bibliothek. Wie durch einen Zufall lautet er *libc.so.5*, also wird das Binary dynamisch gegen die libc.so.5 gebunden.

Nun wird das fertig gebundene Programm aufgerufen:

1. Das Programm ruft den dynamischen Linker `/lib/ld-linux.so.x` auf und teilt ihm mit, daß es Funktionen aus einer dynamischen Bibliothek mit dem so–Namen *libc.so.5* aufzurufen gedenke.
2. `ld-linux` hat jetzt zwei Möglichkeiten: Entweder stellt er fest, daß sich die libc.so.5 bereits im Hauptspeicher befindet, freut sich und gibt die Kontrolle an das aufrufende Programm zurück oder er findet sie noch nicht im Hauptspeicher. Dann schaut er in seinem (mittels `ldconfig` aufgebauten) Cache nach, wo sich eine Datei mit dem so–Namen der Bibliothek wohl befinden möge. Er wird im `/lib`–Verzeichnis fündig und lädt über den vom `ldconfig` angelegten Softlink `/lib/libc.so.5 -> /lib/libc.so.5.4.33` die Bibliothek in den Hauptspeicher, um dann die Kontrolle wieder zurückzugeben.
3. Geschafft! Das Programm kann endlich starten!

[13] Dieser Link wird i.d.R. bei der Installation der libc mit angelegt, er wird aber im Gegensatz zu `libc.so.5 -> libc.so.5.4.33` nicht von einem Programm wie `ldconfig` automatisch erzeugt.

4 Typische Probleme

Für gewöhnlich kommt man am ehesten mit dynamischen Bibliotheken in Berührung, wenn man sich Software z.B. aus dem Internet besorgt hat und versucht, diese zu kompilieren und zu binden oder einfach so zum Laufen zu kriegen.

Ich will hier nun kurz die zwei wichtigsten Probleme, die sich dabei ergeben können darstellen: Fehlende oder nicht gefundene Headerdateien und Bibliotheken.

Grundsätzlich sollte man sich bei solchen Vorhaben an die vorhandene Dokumentation halten: Eine gute Installationsanleitung verrät, welche Bibliotheken und sonstigen Systemkomponenten benötigt werden. Dann kann mittels `/sbin/ldconfig -v` überprüft werden, ob die Bibliotheken bereits vorhanden sind oder ob man sich gleich wieder ins Internet begeben sollte, um die Bibliotheken auch noch downzuloaden.

4.1 Fehlende oder nicht auffindbare Headerdateien

Manchmal ist zwar die benötigte statische oder dynamische Bibliothek im System installiert, es existiert aber keine Headerdatei (im Standardsuchpfad) dazu.

Diese Konstellation ist deshalb nicht so ganz abwegig, weil mit ihr dynamisch gegen die entsprechende Bibliothek gebundene Programme zwar ablaufen können, man kann nur keine solchen Programme selber übersetzen. Manche Distributionen installieren die Headerfiles nur dann, wenn explizit Unterstützung für ein „Entwicklersystem“ angefordert wurde.

Typischerweise treten dabei Fehlermeldungen der folgenden Art auf:

```
# make
gcc -c usehello.c -o usehello.o
usehello.c:2: libhello1.h: No such file or directory
make: *** [usehello.o] Error 1
#
```

Hier gibt es zwei Lösungsmöglichkeiten:

- Wenn man mit `locate` *libname*`.h` die Headerdatei abseits der üblichen Pfade[14] findet, kann man sie in eines der Standardverzeichnisse kopieren oder das Makefile so abändern, daß der Suchpfad mit `-L`*path* entsprechend erweitert wird.
- Findet sich die Headerdatei partout nicht im Filesystem, muß sie entweder aus dem entsprechenden (Versionsnummer beachten!) Paket der Bibliothek aus dem Internet besorgt werden oder gleich eine ganz neue Version der Bibliothek installiert werden.

[14] z.B. `/usr/include` und `/usr/local/include`

4.2 Fehlende oder nicht auffindbare Bibliotheken

Beizeiten ist auch die entsprechende Bibliothek noch gar nicht auf dem Rechner vorhanden bzw. liegt wiederum nicht in einem der Standardverzeichnisse oder ist schlichtweg nicht zur erforderlichen Version kompatibel.

Es kommt dann entweder unmittelbar eine Ausgabe wie

```
/usr/bin/ld: cannot open -lhello: No such file or directory
```

oder man wird mit ungefähr folgenden Fehlermeldungen bedacht:

```
# make
gcc -c usehello.c -o usehello.o
gcc usehello.o -lhello -o usehello_dynamic
usehello.o: In function ‘main’:
usehello.o(.text+0x4): undefined reference to ‘print_hello1’
usehello.o(.text+0x9): undefined reference to ‘print_hello2’
collect2: ld returned 1 exit status
make: *** [usehello_dynamic] Error 1
#
```

Beim ersten Fall konnte die Bibliothek gar nicht erst gefunden werden. Die Abhilfe erfolgt hier wie eben: Mit `locate` die Bibliothek suchen, wenn sie gefunden wurde, kann man sie in ein Standardverzeichnis kopieren oder den Suchpfad erweitern, ansonsten muß man sie neu installieren.

Beim zweiten Fall wurde zwar „etwas“ gefunden, aber diese Bibliothek beinhaltet nicht alle benötigten Funktionen. Abhilfe schafft hier evtl. eine kompatible Version der Bibliothek, vielleicht ist aber auch das Makefile falsch (angepaßt worden), und es fehlt schlichtweg die `-l`*libname*–Option zum Binden der entsprechenden Bibliothek. In diesem Falle sollte die Installationsdokumentation zu Rate gezogen werden.

5 Ein Beispiel

Als Beispiel betrachte ich nun das berühmt berüchtigte „Hello World!“ Programm, einmal ohne Verwendung von Bibliotheken und anschließend zuerst unter Verwendung einer statischen und dann einer dynamischen Bibliothek.

Das Programm besteht aus drei Teilen[15]:

- In den Dateien `libhello1.h` (Header–Datei, in der die Schnittstelle deklariert wird) und `libhello1.c` (Implementierung) ist die Funktion `print_hello1` definiert. Sie gibt den Text „Hello“ aus.
- In den Dateien `libhello2.h` und `libhello2.c` ist die Funktion `print_hello2` definiert. Sie gibt den Text „World!“ aus.
- In der Datei `usehello.c`, die unser Hauptprogramm und damit den Benutzer der Bibliotheken darstellt, ist die Funktion `main` definiert, die beim Programmstart aufgerufen wird. `main` ruft einfach nacheinander die beiden Funktionen `print_hello1` und `print_hello2` auf und beendet sich dann.

Man beachte, daß die Signaturen der beiden `print_hello`–Funktionen durch die `include`–Klauseln im Hauptprogramm dem Compiler bekannt gemacht worden sind.

5.1 Keine Verwendung von Bibliotheken

Zunächst wird das Programm ohne Benutzung von Bibliotheken übersetzt und gebunden:

```
# make ohne
gcc -c libhello1.c -o libhello1.o
gcc -c libhello2.c -o libhello2.o
gcc -c usehello.c -o usehello.o
gcc libhello1.o libhello2.o usehello.o -o usehello_ohne #V1
#
```

Der Befehl `nm -g libhello1.o libhello2.o usehello.o` zeigt deutlich, wie sich das Programm über die einzelnen Dateien verteilt:

```
# nm -g libhello1.o libhello2.o usehello.o
libhello1.o:
00000000 T print_hello1
         U printf
libhello2.o:
00000000 T print_hello2
         U printf
usehello.o:
00000000 T main
         U print_hello1
         U print_hello2
#
```

[15] siehe dazu den Sourcecode im Anhang A

Er zeigt die in Objektdateien definierten externen Symbole an. Hierbei wird ersichtlich, daß `libhello1.o` und `libhello2.o` jeweils ihre `print_hello`–Funktion implementieren (dafür steht das „T“) und die Funktion `printf` noch undefiniert ist („U“); sie muß von der libc zur Verfügung gestellt werden. `usehello.o` implementiert die `main`–Funktion und will beide `print_hello`–Funktionen aufrufen, sie sind jedoch in der Datei selbst nicht implementiert, daher auch hier die „U“s.

`nm -u` zeigt an, welche undefinierten Symbole in einer Datei enthalten sind. Diese müssen also durch andere, statisch oder dynamisch dazugebundene Dateien definiert werden:

```
# nm -u usehello_ohne
__libc_init
__setfpucw
atexit
exit
printf
#
```

Das sind die Funktionen (oder Datenstrukturen), die von der dynamisch gelinkten libc zur Verfügung gestellt werden müssen. Wer dabei aufgepaßt hat, fragt sich vielleicht: „Wo kommen jetzt diese undefinierten Symbole her, die aus keiner der drei Objektdateien stammen?“ Die einfache Antwort: Außer der libc bindet der Linker defaultmäßig auch noch fünf weitere Objektdateien[16] aus dem `/usr/lib`–Verzeichnis zum Executable hinzu. Die weiteren undefinierten Symbole werden von diesen Objektdateien verlangt und befinden sich ebenfalls in der libc.

Der `ldd`–Befehl zeigt, von welchen dynamischen Bibliotheken (so–Name und absoluter Pfad der Datei) ein Executable abhängt, in diesem Falle ist es nur die libc:

```
# ldd usehello_ohne
    libc.so.5 => /lib/libc.so.5
#
```

Und schließlich läuft das Programm sogar:

```
# ./usehello_ohne
Hello World!
#
```

Bei der zweiten Variante kann nach Änderung des Makefiles auch noch die libc statisch zum Executable gebunden werden:

```
# mv usehello_ohne usehello_ohne_v1
# make ohne
gcc -static libhello1.o libhello2.o usehello.o -o usehello_ohne #V2
# mv usehello_ohne usehello_ohne_v2
# ldd usehello_ohne_v2
    statically linked (ELF)
# nm -u usehello_ohne_v2
#
```

[16] Welche das sind, kriegt man ganz leicht mit der `-v`–Option des `gcc` heraus: `/usr/lib/crt1.o`, `/usr/lib/crti.o`, `/usr/lib/crtbegin.o`, `/usr/lib/crtend.o` und `/usr/lib/crtn.o`.

Man beachte insbesondere die Größenunterschiede der beiden erzeugten Executables:

```
# ls -al usehello_ohne_v?
-rwx------ 1 lobyte users   4253 Feb 12 05:32 usehello_ohne_v1
-rwx------ 1 lobyte users 105900 Feb 12 05:57 usehello_ohne_v2
#
```

Hier wird schnell deutlich, wie sinnvoll der Einsatz dynamischer Bibliotheken ist!

5.2 Verwendung von statischen Bibliotheken

Nun will ich die beiden „Module“ `libhello1` und `libhello2` zu einer statischen Bibliothek zusammenfassen:

```
# make static
ar rcs libhello.a libhello1.o libhello2.o
gcc usehello.o -L. -lhello -o usehello_static
#
```

Dieses geschieht mit Hilfe des `ar`–Befehls. Anschließend kann die Bibliothek mit Hilfe des `-l`–Parameters beim Binden mitbenutzt werden.

Mittels `nm -s` kann man nun in die statische Bibliothek hineinschauen:

```
# nm -s libhello.a

Archive index:
print_hello1 in libhello1.o
print_hello2 in libhello2.o

libhello1.o:
00000000 t gcc2_compiled.
00000000 T print_hello1
         U printf

libhello2.o:
00000000 t gcc2_compiled.
00000000 T print_hello2
         U printf
#
```

Rein auf Objektdatei–Ebene geht das natürlich auch wieder mit dem `ar`–Befehl:

```
# file libhello.a
libhello.a: current ar archive
# ar t libhello.a
libhello1.o
libhello2.o
#
```

Das Binary, das dabei herauskommt ist identisch zu dem, was bei Variante 1 ohne Verwendung von Bibliotheken entstand; nur das Kompilieren und Binden gestaltet sich hier anders.

5.3 Verwendung von dynamischen Bibliotheken

Nun aber endlich zur Variante mit einer dynamischen Bibliothek:

```
# make dynamic
gcc -fPIC -c libhello1.c -o libhello1.so
gcc -fPIC -c libhello2.c -o libhello2.so
gcc -shared -Wl,-soname,libhello.so.0 libhello1.so libhello2.so -o libhello.so.0.0
ln -s libhello.so.0 libhello.so
ln -s libhello.so.0.0 libhello.so.0
gcc usehello.o -L. -lhello -o usehello_dynamic
#
```

Zuerst müssen die Objektdateien der Bibliothek mit dem `-fPIC`–Switch erzeugt werden, anschließend können diese mit dem speziellen gcc–Linker–Aufruf zur Bibliothek `libhello.so.0.0` mit dem so–Namen `libhello.so.0` verschmolzen werden. Nachdem zwei Softlinks zur Festlegung der zu verwendenden Bibliotheksversion angelegt worden sind, kann der eigentliche Linker ebenso wie in der Variante für statische Bibliotheken aufgerufen werden.

Nun sehe ich mir die entstandenen Dateien etwas genauer an:

```
# file libhello.so.0.0
libhello.so.0.0: ELF 32-bit LSB dynamic lib i386 (386 and up) Vers. 1
# ldd usehello_dynamic
   libhello.so.0 => not found
   libc.so.5 => /lib/libc.so.5
#
```

Oops, so würde ich das Programm nicht starten können:

```
# ./usehello_dynamic
./usehello_dynamic: can't load library 'libhello.so.0'
#
```

Es fehlt die besagte Umgebungsvariable:

```
# export LD_LIBRARY_PATH=`pwd`
# ldd usehello_dynamic
   libhello.so.0 => /home/lobyte/hfi/edv2/vertiefung/libs/beispiel/libhello.so.0
   libc.so.5 => /lib/libc.so.5
# ./usehello_dynamic
Hello World!
#
```

Wie erwartet ist das dynamisch gebundene Executable auch das kleinste:

```
# ls -al usehello_ohne_v* usehello_static usehello_dynamic
-rwx------ 1 lobyte users 105900 Feb 12 05:57 usehello_ohne_v2
-rwx------ 1 lobyte users   4253 Feb 12 05:32 usehello_ohne_v1
-rwx------ 1 lobyte users   4253 Feb 12 06:40 usehello_static
-rwx------ 1 lobyte users   4010 Feb 12 06:45 usehello_dynamic
#
```


A Der Sourcecode des Beispiels

A.1 usehello.c

```
/* usehello.c */
#include "libhello1.h"
#include "libhello2.h"

int main (void)
{
   print_hello1();
   print_hello2();
   return 0;
}
```

Hierzu noch folgende Anmerkung: Man könnte zwar auch die beiden Headerdateien zu einer zusammenfassen, um *eine* Headerdatei als Schnittstellenspezifikation für die Bibliotheken zu haben, es ist jedoch durchaus nicht unüblich, daß Bibliotheks–Schnittstellen auf mehrere Headerdateien aufgeteilt sind.

A.2 libhello1.h und libhello1.c

```
/* libhello1.h */

void print_hello1 (void);
```

```
/* libhello1.c */
#include <stdio.h>

void print_hello1 (void)
{
   printf ("Hello ");
}
```

A.3 libhello2.h und libhello2.c

```
/* libhello2.h */

void print_hello2 (void);
```

```
/* libhello2.c */
#include <stdio.h>

void print_hello2 (void)
{
   printf ("World!\n");
}
```

A.4 makefile

```
# makefile fuer das Hello World - Bibliotheksbeispiel

# ohne Bibliotheken
# Variante 1: libc noch dynamisch zum Executable gebunden
# Variante 2: libc auch statisch ins Executable gebunden (eher unueblich)

ohne: libhello1.o libhello2.o usehello.o usehello_ohne

libhello1.o: libhello1.h libhello1.c
   gcc -c libhello1.c -o libhello1.o

libhello2.o: libhello2.h libhello2.c
   gcc -c libhello2.c -o libhello2.o

usehello_ohne: libhello1.o libhello2.o usehello.o
   gcc libhello1.o libhello2.o usehello.o -o usehello_ohne #V1
   #gcc -static libhello1.o libhello2.o usehello.o -o usehello_ohne #V2


# statische Bibliothek

static: libhello1.o libhello2.o libhello.a usehello.o usehello_static

libhello.a: libhello1.o libhello2.o
   ar rcs libhello.a libhello1.o libhello2.o

usehello_static: libhello.a usehello.o
   gcc usehello.o -L. -lhello -o usehello_static


# dynamische Bibliothek

dynamic: libhello1.so libhello2.so libhello.so.0.0 usehello.o usehello_dynamic

libhello1.so: libhello1.h libhello1.c
   gcc -fPIC -c libhello1.c -o libhello1.so

libhello2.so: libhello2.h libhello2.c
   gcc -fPIC -c libhello2.c -o libhello2.so

libhello.so.0.0: libhello1.so libhello2.so
   gcc -shared -Wl,-soname,libhello.so.0 libhello1.so libhello2.so \
       -o libhello.so.0.0
   ln -s libhello.so.0 libhello.so
   ln -s libhello.so.0.0 libhello.so.0

usehello_dynamic: libhello.so.0.0 usehello.o
   gcc usehello.o -L. -lhello -o usehello_dynamic
```



```
# Starten des dynamischen gebundenen Binaries

run: usehello_dynamic
   export LD_LIBRARY_PATH=`pwd`; \
   ./usehello_dynamic


# usehello
usehello.o: libhello1.h libhello2.h usehello.c
   gcc -c usehello.c -o usehello.o


clean:
   @rm -f libhello1.o libhello1.so libhello2.o libhello2.so
   @rm -f libhello.a libhello.so libhello.so.0 libhello.so.0.0
   @rm -f usehello.o usehello_ohne usehello_static usehello_dynamic
```

B Vom Sourcecode zum ausführbaren Programm

Hier sollen noch kurz der Vollständigkeit halber alle Schritte vorgestellt werden, die zum Erzeugen eines ausführbaren Programmes aus dem Sourcecode nötig sind. In den Klammern ist der Name des Programmes angegeben, das den jeweiligen Schritt ausführt:

1. Vorübersetzen (Precompiler),
2. Übersetzen (Compiler),
3. Assemblieren (Assembler) und
4. Linken (Linker).

Der „normale“ gcc, mit dem man C–Programme übersetzt, ist nicht direkt der C–Compiler, sondern lediglich ein Frontend, das der Reihe nach die entsprechenden „richtigen“ Programme aufruft. Gibt man dem gcc den Parameter `-v` mit, so zeigt er einem auch jeden einzelnen Schritt an. Anhand dieser Ausgaben vom Hello–World–Beispiel will nun ich die einzelnen Schritte erklären.

Zunächst wird das Frontend gcc mit Parameter `-v` aufgerufen:

```
# gcc -v hello_world.c -o hello_world
Reading specs from /usr/lib/gcc-lib/i486-linux/2.7.2.1/specs
gcc version 2.7.2.1
```

Da der gcc angewiesen wurde, jede seiner einzelnen Aktivitäten genau zu erläutern, erklärt er mir in der ersten Zeile, aus welcher Datei er bestimmte Voreinstellungen einliest. Anschließend begrüßt er mich mit seinem Namen und seiner Versionsnummer.

Wenn man weitere Parameter oder Optionen angibt, erkennt der gcc, für welchen Schritt diese Angabe relevant ist und gibt sie nur an das jeweilige Programm weiter. So wird die Debugoption `-g` z.B. nur an den eigentlichen C–Compiler, *cc1*, weitergeleitet, da sie für die anderen Programme irrelevant ist.

Ein weiteres Feature vom gcc ist, daß man bestimmen kann, bis zu welchem der Schritte er arbeiten soll: Ohne weitere Angaben führt er alle Schritte aus. Bei `-E`, `-S` und `-c` ist nach dem ersten, zweiten bzw. dritten Schritt Schluß.

B.1 Der Precompiler

Der erste Arbeitsschritt, der auszuführen ist, ist der Aufruf des sog. *Precompilers*. Dieser kann mit Statements der Programmiersprache C nicht sonderlich viel anfangen, sondern durchsucht den gesamten Sourcecode nach bestimmten Anweisungen, die nur für ihn gedacht sind (d.h. die mit einem „#“ anfangen) und führt diese aus. Der bekannteste Precompiler–Befehl dürfte dabei `#include` sein: Der *cpp*, so heißt der Precompiler unter Linux, entfernt das include–Statement und ersetzt es durch die entsprechende Datei, die er an genau dieser Stelle einfügt.

```
/usr/lib/gcc-lib/i486-linux/2.7.2.1/cpp -lang-c -v -undef -D__GNUC__=2
-D__GNUC_MINOR__=7 -D__ELF__ -Dunix -Di386 -Dlinux -D__ELF__ -D__unix__
-D__i386__ -D__linux__ -D__unix -D__i386 -D__linux -Asystem(unix)
-Asystem(posix) -Acpu(i386) -Amachine(i386) -D__i486__ hello_world.c
/tmp/cca11442.i
GNU CPP version 2.7.2.1 (i386 Linux/ELF)
#include "..." search starts here:
#include <...> search starts here:
 /usr/local/include
 /usr/i486-linux/include
 /usr/lib/gcc-lib/i486-linux/2.7.2.1/include
 /usr/include
End of search list.
```

In der ersten Zeile finde ich den genauen Befehlsaufruf, den der gcc für mich tätigt. Die vielen `-Dxxx`–Statements definieren verschiedene sog. *Makros*, zum Schluß wird die Sourcecode–Datei und die Zieldatei, hier eine temporäre Datei, angegeben. Auch der cpp grüßt zunächst höflich, bevor er zur Kontrolle seine eingebauten Suchpfade für Include–Dateien ausgibt. Falls der cpp Probleme bekommen sollte, also z.B. eine Headerdatei nicht finden kann, würde hier die entsprechende Fehlermeldung erscheinen. Ergebnis dieses Schrittes ist eine vorkompilierte Datei, in der sämtliche Preprozessor–Anweisungen abgearbeitet worden sind.

B.2 Der Compiler

Nun wird der eigentliche C–Compiler aufgerufen:

```
/usr/lib/gcc-lib/i486-linux/2.7.2.1/cc1 /tmp/cca11442.i -fno-strength-
reduce -quiet -dumpbase hello_world.c -version -o /tmp/cca11442.s
GNU C version 2.7.2.1 (i386 Linux/ELF) compiled by GNU C version 2.7.2.1.
```

Er bekommt als Eingabe die soeben erzeugte, temporäre Datei und liefert eine neue Datei im /tmp–Verzeichnis. Diese beinhaltet nun das kompilierte C–Programm in Form eines Assembler–Listings. In der Assemblersprache werden die CPU–internen, als Zahl kodierten Maschinensprachebefehle in Form von textuellen Kürzeln ausgedrückt. Dieses Ergebnis ist zwar noch halbwegs vom Menschen lesbar, nur leider kann der Mikroprozessor damit noch nichts anfangen.

B.3 Der Assembler

Das Assemblerprogramm muß nun noch in die Maschinensprache der CPU “übersetzt“[17] werden. Das Programm, das diese Aufgabe übernimmt, heißt ebenfalls *Assembler*:

```
/usr/i486-linux/bin/as -V -Qy -o /tmp/cca114421.o /tmp/cca11442.s
GNU assembler version 2.8.1 (i486-linux), using BFD version 2.8.1
```

Er liefert, welch Wunder, eine weitere /tmp–Datei. Das ist jetzt eine Objektdatei, die die hochsprachlichen Konstrukte des ursprünglichen C–Programmes in Maschinensprache kodiert beinhaltet. Sie ist so nur noch nicht lauffähig.

B.4 Der Linker

Erst der Linker macht aus einer oder mehreren Objektdateien ein lauffähiges Programm:

```
/usr/i486-linux/bin/ld -m elf_i386 -L /usr/X11R6/lib -rpath-link
/usr/X11R6/lib -dynamic-linker /lib/ld-linux.so.1 -o hello_world
/usr/lib/crt1.o /usr/lib/crti.o /usr/lib/crtbegin.o -L/usr/lib
/gcc-lib/i486-linux/2.7.2.1 -L/usr/i486-linux/lib /tmp/cca114421.o
-lgcc -lc -lgcc /usr/lib/crtend.o /usr/lib/crtn.o
```

Auch wenn man diesen komplizierten Aufruf sieht, ist man wieder dankbar, daß einem der gcc diese ganze Arbeit abnimmt. Dem Linker wird hier im Einzelnen mitgeteilt, mit welchen (dynamischen) Bibliotheken und weiteren Objektdateien unsere eigene Objektdatei „zusammengebunden“ werden soll. Hier beim Linkeraufruf kommt auch der `-o hello_world`–Parameter an, den ich ganz am Anfang dem gcc mitgegeben hatte: So wird das fertig gebundene und ausführbare Programm `hello_world` erzeugt.

[17] Das Assemblieren ist sozusagen eine Eins–zu–Eins–Übersetzung.