

Prof. L. Thiele

Technische Informatik 1 - HS 2011

Übung 3

Datum: 27.10.2011

Einleitung

Das Ziel dieser Übung ist, den Compilations-Vorgang und insbesondere den Aufbau von Objekt-Dateien kennenzulernen.

Die Übung findet auf einem Linux Rechner mit einer Intel x86 Architektur statt. Um auf diesem Rechner MIPS Programme zu compilieren, verwenden wir einen sogenannten *Cross-Compiler*, der Code für einen fremden Instruktionssatz generieren kann. Die generierten Applikationen können dann auf einem MIPS Prozessor oder auf einem MIPS Prozessor Emulator ausgeführt werden.

Um die Cross-Compilation-Tools für die MIPS Architektur vom Compiler für die x86 Architektur (so genannter Host-Compiler) unterscheiden zu können, tragen alle MIPS Tools einen *mips-elf*-Präfix. Der MIPS-C-Compiler wird folglich mit dem Namen *mips-elf-gcc* aufgerufen. Im Praktikum werden wir hauptsächlich folgende Tools verwenden:

<code>mips-elf-gcc</code>	C-Compiler für die MIPS Architektur
<code>mips-elf-objdump</code>	Liest Informationen aus Objekt-Dateien aus (Segmente, Symboltabelle, Programmcode)
<code>mips-elf-nm</code>	Gibt Symboltabelle von Objekt-Dateien aus
<code>mips-elf-run</code>	MIPS-Emulator für die Ausführung von MIPS Programmen auf dem Host-Prozessor

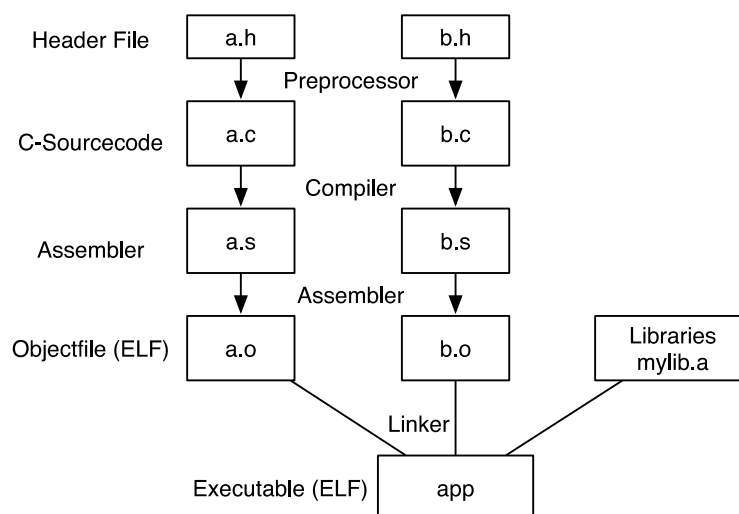


Abbildung 1: Übersicht über den Compilations-Toolflow für ein C Programm

Abbildung 1 präsentiert den Toolflow bei der Compilation eines allgemeinen C Programmes. Die Applikation besteht aus 2 Modulen (C-Quelldateien), die mit dem Compiler in Assembler und dann vom Assembler

in Maschinencode (Objekt-Dateien) übersetzt werden. Der Aufruf des Assemblers erfolgt üblicherweise implizit durch den Compiler. Die Objekt-Dateien beider Module (a.o und b.o) sowie zusätzliche System-Bibliotheken (Libraries) werden vom Linker in ein ausführbares Programm, ein sogenanntes Executable, transformiert.

Objekt-Dateien enthalten nicht nur den Maschinencode des entsprechenden Moduls, sondern auch zusätzliche Daten und Informationen, die für den Linker und für die Ausführung des Programms von Bedeutung sind. Insbesondere besteht eine Objekt-Datei aus folgenden Inhalten:

- Symbole (Abbildung von Funktions- und Variablenamen auf Adressen)
- Ausführbarer Code
- Daten (Speicherbedarf, Adressen und ggf. Initialwerte von globalen Variablen und Konstanten)
- Referenzen auf Funktionen und Daten in anderen Objekt-Dateien
- Relokationsinformationen
- Debuginformationen (z.B. Zusammenhang von Zeilennummern im Quellcode und dem generierten Assembler- bzw. Maschinencode)

Es gibt verschiedene Formate für Objekt-Dateien. In diesem Praktikum verwenden wir das weit verbreitete ELF Format, welches u.a. auch von Linux und Solaris verwendet wird. Beim Linken werden die einzelnen Objekt-Dateien in eine einzige ausführbare Objekt-Datei kombiniert, die ebenfalls im ELF Format gespeichert ist.

1 Anlegung von Objekt-Dateien im Speicher

Eine Objekt-Datei im ELF Format ist in verschiedene Teile gegliedert, die als Sektionen bezeichnet werden. Einige wichtige Sektionen sind in Tabelle 1 zusammengefasst.

Sektion	Inhalt
.text	Programmcode in Maschinsprache.
.rodata	Programmdaten, die initialisiert und zur Laufzeit nicht veränderbar sind (Konstanten).
.data	Programmdaten, die initialisiert und zur Laufzeit vom Programm veränderbar sind (Variablen).
.bss	Nicht initialisierte oder mit 0 initialisierte Variablen. Alle Variablen dieser Sektion werden vor der Ausführung automatisch mit 0 initialisiert.

Tabelle 1: Sektionen einer Objekt-Datei im ELF Format (unvollständig)

a) Nehmen Sie an, eine Applikation soll auf einem eingebetteten Mikroprozessorsystem ausgeführt werden, das auf einer MIPS CPU beruht und RAM sowie (Flash-)ROM Speicherbausteine besitzt. Geben Sie für jede der in Tabelle 1 genannten Sektionen an, in welchem Speichertyp der jeweilige Inhalt angelegt werden soll:

Sektion	RAM	ROM
.text	<input type="checkbox"/>	<input type="checkbox"/>
.rodata	<input type="checkbox"/>	<input type="checkbox"/>
.data	<input type="checkbox"/>	<input type="checkbox"/>
.bss	<input type="checkbox"/>	<input type="checkbox"/>

- b) Nehmen Sie an, dass zum Zeitpunkt des Bootens des eingebetteten Systems alle Sektionen im ROM gespeichert sind. Überlegen Sie, wie der Bootvorgang funktionieren muss und notieren Sie den zeitlichen Ablauf. Geben Sie dabei an, wie die einzelnen Sektionen behandelt werden müssen (z.B. wie und wann die Daten im RAM initialisiert werden).

2 Toolflow

In dieser Aufgabe untersuchen wir zunächst, wie der Compiler aus einem C-Programm Assemblercode generiert. Anschliessend betrachten wir die Generierung von Objekt-Dateien und ausführbaren Dateien.

2.1 Assemblercode

Gegeben ist das Beispielprogramm `sorter.c`, welches den Bubble-sort Sortieralgorithmus implementiert. Erzeugen Sie den entsprechenden MIPS Assemblercode mit Hilfe des folgenden Befehls:

```
> mips-elf-gcc -S -G0 sorter.c
```

Die Option `-S` bewirkt, dass das Programm compiliert aber nicht assembliert wird. Die Option `-G0` vermeidet für diese Übung unerwünschte Optimierungen.

Der in `sorter.s` generierte Assemblercode enthält neben den eigentlichen Instruktionen viele Assemblerdirektiven, die den Assembler bei der Generierung der Objekt-Datei steuern. Einige wichtige Assemblerdirektiven sind in Tabelle 2 zusammengefasst.

Direktive	Auswirkung
<code>.globl</code>	Exportiert ein Symbol, d.h. den Namen einer Funktion oder Variable. Dadurch werden der Name und die Adresse des Symbols beim Compilieren in der Symboltabelle der Objekt-Datei eingetragen.
<code>.rdata</code>	Die folgenden Daten werden in der Sektion <code>.rodata</code> angelegt.
<code>.data</code>	Die folgenden Daten werden in der Sektion <code>.data</code> angelegt.
<code>.bss</code>	Die folgenden Daten werden in der Sektion <code>.bss</code> angelegt.
<code>.text</code>	Die folgenden Instruktionen werden in der Sektion <code>.text</code> (Programmcode) angelegt.
<code>.word</code>	Die angegebenen Werte werden in aufeinanderfolgende Speicherzellen geschrieben. Die Werte werden als 32 bit Zahlen interpretiert. (Diese Direktive wird zur Initialisierung von Datenstrukturen verwendet.)
<code>.align <i>integer</i></code>	Die folgenden Daten werden in der Objekt-Datei Modulo <i>integer</i> Bytes ausgerichtet.

Tabelle 2: MIPS Assemblerdirektiven (unvollständig)

- a) Deklarieren Sie in `sorter.c` zwei globale Integer Variablen a , b und eine globale Integer Konstante c . Initialisieren Sie diese Daten wie folgt: $a = 7$, $b = 0$, $c = 33$. Compilieren Sie das Programm erneut wie oben beschrieben und untersuchen Sie die generierten Assemblerdirektiven. In welchen Sektionen der Objekt-Datei legt der Assembler a , b und c an? Welchen Sinn macht die unterschiedliche Behandlung von a und b ? Welchen Sinn macht die unterschiedliche Behandlung von a und c ?

- b) Deklarieren Sie in der Funktion `main` eine lokale Integer Variable `d` und initialisieren Sie diese mit einem beliebigen Wert. Compilieren Sie das Programm erneut wie oben beschrieben und untersuchen Sie den generierten Assemblercode. Warum wird `d` nicht in einer Daten-Sektion der Objekt-Datei angelegt?
- c) Trennen Sie die Funktionalität des C-Programmes in zwei Module. Lagern Sie hierfür den Quellcode zum Vertauschen zweier angrenzender Array-Elemente (siehe entsprechenden Kommentar in der Funktion `sort`) in eine eigene Funktion `swap` aus, die in einem separaten Modul `swap.c` erstellt werden soll. Zur Vereinfachung der Aufgabe ist eine entsprechende Header-Datei `swap.h` gegeben und bereits in `sorter.c` eingebunden (Achten Sie auf die Kompatibilität von `swap.h` und `swap.c`). Compilieren Sie die beiden Module wie oben beschrieben und untersuchen Sie den generierten Assemblercode. Wie wird in `swap.s` die Funktion `swap` als solche definiert? Wie wird der Aufruf von `swap` in `sorter.s` behandelt?

2.2 Objekt-Dateien und ausführbare Datei

Generieren Sie nun ein ausführbares Programm (Executable) aus den Modulen `swap.c` und `sorter.c`:

```
> mips-elf-gcc -c -G0 swap.c
> mips-elf-gcc -c -G0 sorter.c
> mips-elf-gcc -o app swap.o sorter.o -T idt.ld
```

Die ersten beiden Befehle übersetzen die beiden Quell-Dateien in entsprechende Objekt-Dateien `swap.o` und `sorter.o`. Der letzte Befehl ruft den Linker auf, der die Objekt-Dateien in eine ausführbare Datei `app` kombiniert. Mit der Option `-T` wird dem Linker ein Linker-Skript übergeben, welches die Anordnung der Sektionen im Executable definiert und bei Bedarf auch dafür sorgt, dass Standard-Bibliotheken in das Executable gelinkt werden.¹

- a) Betrachten Sie die Symboltabelle der Objekt-Datei `sorter.o`:

```
> mips-elf-nm sorter.o
```

Welche Symbole sind definiert, welche undefiniert? Was ist der Unterschied zwischen den definierten und undefinierten Symbolen einer Objekt-Datei? (Die Angaben in der ausgegebenen Symboltabelle können in der Dokumentation von `mips-elf-nm` nachgeschlagen werden: `man mips-elf-nm`)

- b) Extrahieren Sie den Programmcode aus der Objekt-Datei `sorter.o`:

```
> mips-elf-objdump -d sorter.o > sorter.o.text.txt
```

Die Option `-d` steht für 'disassemble'. Sie sorgt dafür, dass der Programmcode nicht in binärer Maschinsprache, sondern in Assembler angezeigt wird. Da `objdump` viel Output generiert, leiten wir diesen in die Datei `sorter.o.text.txt` um. Betrachten Sie den Aufruf der Funktionen `swap` und `print_array` im extrahierten Programmcode (Tipp: Finden Sie die Funktionsaufrufe durch Vergleich mit `sorter.c`). Welche Adressen werden als Sprungziele in den entsprechenden `jal` Instruktionen verwendet? Würde das Programm in dieser Form funktionieren? Was muss der Linker tun, um die Funktionsaufrufe zu 'reparieren'?

- c) Extrahieren Sie die Relokations-Informationen aus der Objekt-Datei `sorter.o`:

```
> mips-elf-objdump -r sorter.o > sorter.o.reloc.txt
```

Betrachten Sie die extrahierte Relokationstabelle. Wozu benötigt sie der Linker beim Erstellen des ausführbaren Programms?

- d) Extrahieren Sie den Programmcode aus dem Executable `app`:

```
> mips-elf-objdump -d app > app.text.txt
```

¹Bei Verwendung des üblichen `gcc` Compilers für x86 Prozessoren wäre an dieser Stelle die Angabe eines Linker-Skripts überflüssig, da `gcc` auf vordefinierte Linker-Einstellungen zurückgreifen kann.

Betrachten Sie erneut den Aufruf der Funktionen `swap` und `print_array` im extrahierten Programmcode. Welche Adressen werden jetzt als Sprungziele in den entsprechenden `jal` Instruktionen verwendet? Rekonstruieren Sie die zwei grundlegenden Aktionen des Linkers.

- e) Weshalb ist die Datei `app` viel grösser als die summierten Dateigrössen von `swap.o` und `sorter.o`? Vergleichen Sie zur Beantwortung der Frage den disassemblierten Programmcode von `sorter.o` und `app`.

3 Analyse eines unbekanntes Programmes mit `objdump`

Gegeben sei ein Programm, das einen String mit einer unbekanntes Funktion verschlüsselt. Leider ist der Sourcecode des Programms verloren gegangen. Bekannt ist lediglich, dass jedes Zeichen des Strings einzeln verschlüsselt wird.

Versuchen Sie den Verschlüsselungsalgorithmus aus dem Executable `encrypt` zu rekonstruieren. Kennen Sie den Verschlüsselungsalgorithmus?

Hinweise:

- Benutzen Sie das Programm `mips-elf-objdump` mit geeigneten Parametern.
- Suchen Sie die Verschlüsselungsfunktion.
- Beachten Sie die genaue Semantik der MIPS-Instruktionen `addiu` und `sltiu`.

`addiu`: Beide Operanden werden wie bei `addi` mit dem jeweiligen Vorzeichen interpretiert (0xxxx positive Zahl, 1xxxx negative Zahl dargestellt im Zweierkomplement), jedoch wird bei `addiu`, im Gegensatz zu `addi`, im Falle eines Overflows keine Exception gemeldet.

`sltiu`: Beide Operanden werden, im Gegensatz zu `slti`, vorzeichenlos interpretiert. Das bedeutet, dass für `sltiu` Zahlen mit 1xxxx grösser sind als Zahlen mit 0xxxx).

- Zur Lösung dieser Aufgabe ist eine ASCII Tabelle hilfreich (z.B. <http://www.lookupables.com>).