Die Entwicklungsumgebung

Hello World

Das folgende bekannte Programm können wir etwa in der Datei `hello.c` ablegen.

```
#include <stdio.h>

int main() {
   printf("hello world\n");
   return 0;
}
```

Wenn wir es mit der folgenden Anweisung auf dem Host übersetzen, erhalten wir eine ausführbare Datei namens `hello`:

```
gcc hello.c -o hello
```

Noch ein Paket

Dazu benötigen wir mindestens den C-Compiler `gcc`. Dieser (und weitere Werkzeuge) sind im Paket `gcc` enthalten.

Nicht ganz so einfach

Die Ausführung von `hello` schlägt zunächst fehl:

```
hello
```

Das Betriebssystem findet `hello` nicht.

Lösung 1:

Angabe des Verzeichnisses:

```
./hello
```

liefert das gewünschte Ergebnis:

```
hello world
```

Umgebungsvariable

Lösung 2:

Das Betriebssystem durchsucht alle Verzeichnisse, die in der **Umgebungsvariablen** `PATH` definiert sind nach ausführbaren Dateien. Mit dem Befehl `env` sehen wir die Inhalte **aller** Umgebungsvariablen.

`PATH` enthält nicht das aktuelle Verzeichnis.

Das können wir ändern: `PATH=$PATH:.`

- Mit `:` werden Verzeichnisse voneinander getrennt
- `.` bezeichnet das aktuelle Verzeichnis

Die Anweisung hängt also `.` an den bestehenden Pfad an.
Achtung: Immer an `PATH` anhängen, nicht ersetzen. Sonst gehen wichtige Einstellungen verloren.

Variable bei der Anmeldung setzen

Auf Dauer ist es lästig, nach jedem Anmelden die Umgebung erneut einzustellen.

Lösung:

Beim Anmelden eines Anwenders werden automatisch mehrere Skripte ausgeführt. Die Namen dieser Skripte hängen von der Linux-Distribution und der verwendeten Shell ab. Diese Skripte beginnen aber immer mit `.`

Problem:

Der Befehl `ls` zeigt Dateien, die mit `.` anfangen nicht an.

Alle Dateien sieht man wie folgt:

```
ls -lisa
```

Variable bei der Anmeldung setzen

Auf unserem System gibt es mehrere dieser Skripte:
Beispiele: `.bashrc` oder `.profile`

Wir tragen in die letzte Zeile `.bashrc` die folgende Zeile ein:

```
export PATH=$PATH:.
```

Wir starten eine neue Shell und prüfen mit dem Befehl `env`, ob die Umgebung richtig gesetzt wurde.

Achtung:

Das Betriebssystem und viele Werkzeuge nutzen Umgebungsvariable *intensiv*. Das Setzen der richtigen Umgebung erspart uns einige Arbeit!

Den Export nicht vergessen

Gelegentlich definiert man Umgebungsvariable, vergisst aber sie zu exportieren.

Sie werden dann **nicht** an aufgerufene Programme weitergegeben!

Mit

```
echo $hello
```

wird der Wert der Umgebungsvariablen `hello` angezeigt, auch wenn sie nicht exportiert ist.

Mit

```
env | grep hello
```

wird die Variable `hello` nur angezeigt, wenn sie exportiert ist.

...nicht auf dem Target

Die Datei `hello` können wir jetzt auf dem Host, aber nicht auf dem Target ausführen.

Der Grund wird uns klar, wenn wir den Inhalt mit dem Befehl `file` analysieren:

```
hello: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses
shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=0xc5617aa0d9e34c0f9ec9744577ff2f39
9b524bc4, not stripped
```

Die Toolchain

- Eine Sammlung von Werkzeugen (wie Compiler, Linker und Assembler), die man zur Entwicklung von Software benötigt, wird oft als Toolchain bezeichnet. Das Paket `gcc` liefert uns eine solche Toolchain für **Intel**-Prozessoren.
- Das Target hat in vielen Fällen nicht genügend **Ressourcen**, um komplexere Software-Pakete zu übersetzen.
- Wir benötigen eine Toolchain, mit der wir **auf einem Intel-Host** Software **für** ARM-Prozessoren entwickeln können.

Die Toolchain

- Grundsätzlich ist eine Toolchain im Paket `gcc-arm-linux-gnueabihf` vorhanden.
- Es fehlt allerdings der Debugger, der später gebraucht wird.
- Eine geeignete Toolchain erhalten wir bei Linaro. Sie wird - den Unix-Konventionen folgend - im Verzeichnis `/opt` abgelegt.

Die richtige Umgebung

Wenn wir die passenden Umgebungsvariablen für die Toolchain auf dem Host setzen, spart uns das später einige Zeit:

```
ARCH=arm
CROSS_COMPILE=arm-linux-gnueabihf-
CC=${CROSS_COMPILE}gcc
```

- `ARCH` enthält die Bezeichnung der Prozessorarchitektur des Targets
- `CROSS_COMPILE` bezeichnet das gemeinsame Präfix aller Werkzeuge der Toolchain.
- `CC` enthält speziell den Namen des so genannten Cross-Compilers, der Code für ARM-Prozessoren erzeugt.

Übersetzen für das Target

Wir übersetzen `hello.c` jetzt mit dem Cross-Compiler:

```
$CC hello.c -o hello
```

...und können die Datei auf dem Target ausführen.

Mehrere Quelldateien

Wir fügen jetzt zwei C-Quelldateien hinzu:

1. `add.c`

```
int add(int i){
    return i+1;
}
```

2. `multiply.c`

```
int multiply(int i){
    return 2*i;
}
```

`hello` ändern

...und ergänzen `hello.c` um eine Zeile, die die beiden neuen Funktionen aufruft:

```
int main() {
    printf("hello world\n");
    printf("result %i\n",add(multiply(1)));
    return 0;
}
```

Lange Compileraufrufe

Übersetzt wird wie folgt:

```
$CC hello.c add.c multiply.c   -o hello
```

Im Laufe der Zeit kommen viele (möglicherweise mehrere hunderte) Quelldateien zusammen. Es ergeben sich zwei Probleme:

1. Die Aufrufe des Compilers werden länger, unübersichtlicher und fehleranfälliger
2. Es wird immer **alles** übersetzt - und das kann dauern.

make

Das Werkzeug **make** erleichtert unsere Arbeit erheblich:

1. Wir können in einer Datei konfigurieren, **welche Dateien** betroffen sind.

2. Wir können mit Optionen spezifizieren, **welche Operationen** ausgeführt werden sollen

3. Wir können Abhängigkeiten definieren. Es wird dann nur **das Nötigste** übersetzt.

Aufgerufen wird wie folgt:

```
make
```

Die Regeln werden standardmäßig in einer Datei namens **Makefile** definiert.

Ein Makefile

Ein Beispiel für ein Makefile:

```
all: hello
clean:
        -rm -f hello
rebuild: clean all
hello: hello.o multiply.o add.o
        ${CC} hello.o add.o multiply.o -o hello
hello.o: hello.c
        ${CC} -c hello.c
add.o: add.c
        ${CC} -c add.c
multiply.o: multiply.c
        ${CC} -c multiply.c
```

Beispiel:

Beim Aufruf von `make rebuild` wird zunächst `hello` gelöscht und dann der für die Übersetzung von `hello` erforderliche Compileraufruf abgesetzt.

Der Compiler wird nur aufgerufen, wenn sich mindestens eine der Dateien `hello.c`, `add.c` oder `multiply.c` **geändert** hat.

Nur das Nötigste

Der Aufruf

```
make rebuild
```

übersetzt alles. Wird `make` unmittelbar darauf erneut aufgerufen, ergibt sich:

```
make: Nothing to be done for `all'.
```

Es gab ja schließlich keine Änderungen! Von `make` werden nur die unbedingt notwendigen Aufgaben ausgeführt.

Achtung:

- Makefiles können sehr komplex werden.
- Für Makefiles gibt es eine eigene Syntax.

Statisches Binden

Der Compiler-Aufruf mit mehreren Quelldateien, den wir gesehen haben, wird auch als statisches Binden bezeichnet:

Es wird eine **einzige** Binärdatei erzeugt.

Oft möchte man aber die Funktionen von C-Quellen als so genannte **Shared Libraries** zusammenfassen und öffentlich zur Verfügung stellen, damit sie von mehreren Programmen genutzt werden können.

Shared Libraries

Der Aufruf

```
$CC -fPIC -c add.c multiply.c
```

erzeugt die beiden Dateien `add.o` und `multiply.o`.

- Mit `fPIC` erzeugen wir ‚Position Independent Code'. Eine notwendige Voraussetzung für Shared Libraries.

Die beiden Objektdateien fassen wir jetzt zur Bibliothek `libhello.o` zusammen:

```
$CC -shared -Wl,-soname,libhello.so -o libhello.so *.o
```

Shared Libraries

Der Aufruf

```
file libhello.so
```

liefert uns den Beleg dafür, dass wir eine Shared Library erzeugt haben:

```
libhello.so: ELF 32-bit LSB shared object, ARM,
version 1 (SYSV), dynamically linked,
BuildID[sha1]=0xb767a7bc7eb5cf963892cd5e8bc2ab2d
1856f4d2, not stripped
```

Bibliotheken einbinden

Die Bibliothek enthält (natürlich) nicht die main-Funktion aus `hello.c`. Wir übersetzen und binden in `hello` die Bibliothek mit ein:

```
$CC hello.c libhello.so -o hello
```

...noch nicht ganz

Wenn wir die ausführbare Datei auf dem Target ausführen, erhalten wir

```
./hello: error while loading shared libraries:
libhello.so: cannot open shared library
```

Die Bibliothek ist zwar vorhanden, wird aber **nicht gefunden**.

Für Shared Libraries gibt es -ähnlich wie `PATH`- die Umgebungsvariable `LD_LIBRARY_PATH`, die wir richtig setzen müssen:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/mnt/XXX
```

Jetzt wird die Bibliothek gefunden und `hello` wird ausgeführt.

Welche Bibliotheken werden benutzt?

Achtung:

Man muss häufiger nach fehlenden Bibliotheken suchen. Es ist also nützlich zu wissen, welche Shared-Library ein Programm verwendet. Dazu gibt es das Werkzeug **ldd**.
Wenn der **LD_LIBRARY_PATH nicht** richtig gesetzt ist erhalten wir

```
ldd hello

libhello.so => not found
libc.so.6 => /lib/arm-linux-gnueabihf/libc.so.6
(0xb6e7d000)
/lib/ld-linux-armhf.so.3 (0xb6f67000)
```

Der Detektiv

Fehlende Bibliotheken sucht man mit **`find`**:

```
find / -name libhello.so
```