Hardwarenahe Softwareentwicklung Tutorium

Makefiles und die Shell

SS 16

Tobias Trabelsi

2. Mai 2016

Hochschule Bochum
Bochum University
of Applied Sciences

Inhaltsverzeichnis

1 Motivation

Um einen Quelltext zu compilieren brauchen wir bei kleineren Projekten nicht viel. Eine Quelltext-Datei, einen Compiler der die ganze Arbeit macht und damit sind wir fertig.
Bei größeren Projekten reicht dies nur leider nicht mehr aus. Sehr schnell brauchen wir noch Header-Dateien und Libraries gegen die wir mit unserem Programm linken wollen oder haben vielleicht unseren Quelltext über verschiedene Verzeichnisse und Unterverzeichnisse verteilt. Um nun nicht endlose Verkettungen von Befehlen in die Bash hacken zu müssen, wurde das Konstrukt make geschaffen.
Make nimmt eine gegebene Konfigurationsdatei, nach deren Regeln und Vorschriften Befehle generiert werden, die auch große Projekte in der richtigen Reihenfolge in Maschienensprache übersetzen.

2 Aufbau von Makefiles

Der Aufbau von Makefiles ist in den wesentlichen Schritten immer identisch

```
1 target: dependencies
2 [tab] system command
```

Ein wichtiger Punkt zum Aufbau von Makefiles ist unter anderem die Namensgebung. Mit dem absetzen des make Befehls auf der Kommandozeile (bash) sucht das Programm nach einer gegebenen Konfigurationsdatei mit dem Namen

Makefile

Wird diese Datei nicht gefunden, gibt das Programm die Meldung:

make: *** No targets specified and no makefile found. Stop.

aus und beendet sich.

2.1 Andere Namensgebung

Natürlich kann eine Makefile auch von diesem Standart abweichen. In diesem Fall muss die Datei allerdings explizit mit der Form:

make -f MyMakefile

angegeben werden.

2.2 Variablen

In Makefiles können Variablen in bash-Notation angegeben und ausgewertet werden. Dies sieht in der Praxis wie folgt aus:

```
1  PROGRAMM = name
2  SOURCES.c = all.c sources.c
3  SOURCES.h = all.h header.h
4
5  SOURCES = \
6                  $(SOURCES.c) \
7                  $(SOURCES.h)
8
9  OBJECTS = \
10                 $(SOURCES.c:%.c%=%.o)
11
12 CC = compiler
13 CFLAGS = -flag -of -desire
14 LDLIBS = -other -flags
15
16 all:     $(PROGRAMM)
17
18 $(PROGRAMM): $(SOURCES.c) $(OBJECTS)
19         $(CC) $(CFLAGS) -o $@ $(OBJECTS) $(LDLIBS)
20
21 clean:
22         rm -rf $(PROGRAMM)
23
24 run:
25         ./name
26
27 war:
28         @echo "make: *** No idea how to make war. War is stupid.  Stop."
29
30 love:
31         @echo "make: *** No rule on how to make \'love'.  Stop."
```

3 Aufgaben - Makefiles

Nehmen Sie folgenden Quelltext als Basis:

```
1 /* Hello World program */
2
3 #include <stdio.h>
4
5 main()
6 {
7     printf("Hello World");
8
9 }
```

3.1 Aufgabe 1

Schreiben Sie nun eine einfache Makefile, welche den Quelltext beim Aufrufen des make Befehls übersetzt und die executable in „hello “umbenennt.

3.2 Aufgabe 2

Erweitern Sie Ihre Makefile um eine „clean “section, welche die erzeugten executables bei aufrufen des „make clean “Befehls wieder entfernt.

3.3 Aufgabe 3

Schreiben Sie nun ein neues Programm, welches die mathematischen Grundrechenarten mit 2 Integern implementiert.

Nutzen Sie zum Aufrufen Ihrer Methoden die main Funktion in der oben beschriebenen Datei.

3.4 Aufgabe 4

Erweitern Sie Ihre Makefile erneut um Ihr neu geschiebenes Programm.

3.5 Aufgabe 5

Erweitern Sie Ihre Makefile ein letztes mal um eine Statusausgabe zu dem jeweiligen Schritt, der gerade ausgeführt wird.
Hierzu kann echo benutzt werden.

4 Linux und die Shell

UNIX ist das erste moderne Mehrbenutzerbetriebssystem und Linux ist neben OS X die populärste Weiterentwicklung von UNIX. Beide haben mit ihren Wurzeln noch sehr viele Gemeinsamkeiten.
UNIX wurde von Programmierern für Programmierern geschrieben. Dies sieht man sehr schnell, wenn man sich die Grundbausteine des Kommando-Prozessors von UNIX ansieht: Der Shell
Fast alle Elemente der Shell versuchen für den Anwender möglichst bequem zu sein und einem Muster zu folgen. Desweiteren wird versucht die benötigten Zeichen auf ein Minimum zu reduzieren.

4.1 Beispiele

Worte, deren Bedeutung auch in allgemeingültigen Abkürzungen ausgedrückt werden können, werden auch abgekürzt.

```
1 cp file dir/designation
```

Listing 4.1: cp statt copy

Wildcards werden zum Zusammenfassen vieler gleicher Elemente unterstützt. Ebenso ist RegEx von vielen UNIX Programmen unterstützt.

```
1 rm *.c
```

Listing 4.2: alle .c dateien löschen

Eines der mächtigsten Elemente in der Shell, ist die so genannte Pipe |. Sie kann zur Verkettung mehrerer Programme genutzt werden und leitet die Ausgabe eines Programmes in

die Eingabe eines anderen weiter.

```
1 ls -al | grep *.h
```

Listing 4.3: zeige nur Header Dateien an

4.2 Bash

In UNIX gab es am Anfang nur eine Shell, die nach Stephen Bourne benannte Bourne Shell, welche üblicherweise unter /bin/sh zu finden ist.
Unter diesem Namen liegen allerdings in den aktuelleren Linux Distributionen leistungsfähigere Shells, welche aber kompatibel zum Original sind.
Die wichtigsten sind:

- Korn-Shell
  - entwickelt von David G. Korn
  - /bin/ksh
- Bourne Again Shell (bash)
  - inzwischen Standard auf Linux
  - entwickelt von Brian Fox
  - /bin/bash
- C-Shell
  - entwickelt University of California Berkeley
  - /bin/csh
- TEXNEX-C-Shell
  - verwendung auf kommerziellen Systemen
  - /bin/tcsh oder /bin/csh

Wie erwähnt ist die /bin/bash inzwischen die Standard Shell unter Linux.
Die Bash, wie auch jede andere Shell lässt sich leicht erweitern. So kann mit dem Namen eines Programms oder Scripts, welches unter /bin zu finden ist, dieses ausgeführt werden. Sollten Übergabeparameter von diesem Programm gefordert werden, die nicht bekannt sind, kann man mit dem Befehl:

man <COMMAND>

Die Anleitung (eng. manual) zu dem entsprechenden Programm aufrufen.

4.3 Wichtige Bash Programme

Um sich in einem Linux Filesystem zurecht zu finden benötigt man eine kleine Anzahl an Befehlen.

- ls
  - list
  - zeigt Dateien und Ordner in einem Verzeichniss an
- cd
  - change directory
  - wechselt von einem Verzeichniss in ein anderes
- pwd
  - print working directory
  - gibt das aktuelle Verzeichniss aus
- cp
  - copy

  - kopiert eine Datei von A nach B

- mv
  - move
  - verschiebt eine Datei von A nach B
- rm
  - remove
  - löscht eine Datei
- cat
  - keine genaue Wortdefinition
  - gibt den Innhalt einer Datei im Terminal aus
- mkdir
  - make directory
  - erstellt ein neues Verzeichniss

5 Aufgaben - Bash

5.1 Aufgabe 1

Öffnen Sie Ihr Terminal und erstellen Sie ein neues Verzeichniss namens „Workspace “.

5.2 Aufgabe 2

Wechseln Sie in das von Ihnen angelegte Verzeichniss.
Kopieren oder Verschieben Sie nun den von Ihnen Programmierten Quellcode, inklusive Makefile in dieses Verzeichniss.

5.3 Aufgabe 3

Lassen Sie sich Informationen über den Compiler gcc und über den aktuellen Kernel ausgeben.

5.4 Aufgabe 4

Lassen Sie sich die Shell aller Benutzer ausgeben.

5.5 Aufgabe 5

Erstellen Sie in Ihrem Ordner ein neues Verzwichniss, welches mit einem . im Namen beginnt.
Lassen Sie sich den Innhalt des aktuellen Verzeichnisses ausgeben.
Was stellen Sie fest?

5.6 Aufgabe 6

Erstellen Sie eine Liste aller Dateien in Ihrem Workspace und speichern Sie diese Liste in einer neuen Datei Namens „index.txt“

5.7 Aufgabe 7

Löschen Sie nun das von Ihnen angelegte Verzeichniss, welches mit einem . im Namen beginnt.