make

Ausarbeitung zum Unix-Tools Proseminar

by Markus Graßl grassl@in.tum.de

Inhaltsverzeichnis:

1. Einfuehrung

Das Tool make ist ein plattformunabhaengiges Programm, dass das Compilieren von Dateien vereinfacht. Es handelt sich hierbei nicht um einen Compiler, sondern lediglich um ein Tool, dem man mitteilt, wie und mit welchen Mitteln ein Programm compiliert werden soll. Daher ist es vielseitig und programmiersprachenunabhaengig einsetzbar.
Es vereinfacht das Compilieren von großen Programmen erheblich, da nicht jedes einzelne Programmteil eigens compiliert werden muss.
Außerdem sorgt make dafür, dass nur die notwendigen Anbhaengigkeiten aktualisiert werden. Dadurch spart man besonders bei großen Projekten viel Rechenleistung und Zeit.

2. Aufbau eines makefiles

2.1 Allgemeines

Ein makefile ist eigentlich sehr einfach aufgebaut. Es besteht aus gleich aufgebauten Tripeln. Diese Tripel bestehenja aus einem Ziele, dessen Abhaengigkeit(en) und den dazugehoerigen Befehlen.

2.2 Ziele

Das Ziel des makefiles gibt an, was aus den angegebenen Abhaengigkeiten erzeugt werden soll. Hier wird angegeben, wie das “Output-File” heißen soll, und welchen Datentyp dieses Ziel haben soll.

2.3 Abhaengigkeiten

In den Abhaengigkeiten wird angegeben, von welchen Quelldateien die Zieldatei abhaengt. Sie stehen in der gleichen Zeile wie das Ziel und werden von diesem mittels eines Doppelpunktes getennt. Falls ein Ziel mehrere Abhaengigkeiten hat, so werden diese nur durch ein Leerzeichen getrennt hintereinander geschrieben.

2.4 Befehle

Die Befehle eines makefiles geben an, welche Aktionen im Falle einer erfuellten Abhaengigkeit durchgefuehrt werden soll. Es wird hier z.B. angegeben, welcher Compiler verwendet werden soll, und mit welchen Optionen er aufgerufen wird.
Die Befehle stehen in den Zeile(n) unter dem Ziel und der Abhaengigkeit. Damit das make weiß, welche Befehle zu einem Ziel gehoeren, und wo eine neues Ziel beginnt, ist es noetig jeden Befehl mittels eines Tabulators, und nicht mittels Leerzeichen, einzuruecken.
Außerdem muß jeder Befehl in einer eigenen Zeile stehen. Damit bei langen Befehlen nicht die Uebersicht verloren geht, ist es jedoch moeglich Zeilen umzubrechen. Hierfuer wird am Ende der umzubrechenden Zeile ein Backslash (“ \ ”) eingefuegt. Nun kann man den Befehl in der naechsten Zeile fortsetzen.

2.5 Beispiel

```
main.o: main.c defs.h
        gcc  -c main.c
```

```
Ziel: Abhaengigkeiten
      Befehl
```

3. Inhalt eines makefiles

3.1Allgemeines

Der Inhalt eines makefiles besteht aus den oben angesprochenen Tripeln sowie weiteren Anweisungen und Kommentaren. Die Tripel koennen entweder als explizite und/oder implizite Regeln angegeben werden.

3.2 Explizite Regeln

Explizite Regeln werden, vor allem in kleineren makefiles, haeufig verwendet. In einer expliziten Regel gibt man fuer genau eine Datei an, von welchen anderen Dateien sie abhaengt, also welche Dateien benoetigt werden, um das Ziel zu erstellen. Es wird in einer expliziten Regel genau angegeben, mittels welcher Befehle sie aus den Quelldateien erstellt wird. Man gibt hier direkt an, welcher Compiler verwendet werden soll und mit welchen

Optionen er verwendet wird.

3.3 Implizite Regeln

Bei einer impliziten Regel gibt man nicht explizit an, was und wie etwas erstellt wird, und von was es abhaengt. Implizite Regels sind eher als eine Art Schablone zu verstehen, mittels derer man alle Dateien, die identische Erstellensregeln haben, compilieren kann. Der Vorteil hierbei liegt klar auf der Hand, gerade bei großen Projekten gibt es haeufig viele Dateien, die die gleiche Endung haben und mittels der gleichen Regel erstellt werden. Hier waere es fatal, wenn es notwendig waere diese Regeln alle einzeln als explizite Regeln auszuformulieren. Statt dessen schreibt man einmal eine implizite Regel, welche dann automatisch fuer alle Dateien des gewuenschten Typs angewandt wird. Viele dieser impliziten Regeln sind bereits in make vordefiniert. D.h. fuer die meißten Regeln ist es nicht einmal notwendig sich solch eine Regel zu definieren, sondern man kann einfach die vordefinierte verwenden.

Um sich eigene implizite Regeln zu definieren, benutzt man sog. “pattern rules”. In diesen gibt man an, welches Ziel aus den Quellen erstellt werden soll. Um eine “pattern rule” zu definieren, ersetzt man einfach den Anfang von Ziel und Abhaengigkeiten in einer Regel durch ein Prozent-Zeichen. Man laeßt lediglich die Endungen bestehen. So erhaelt man eine implizite Regel, die nun universell auf alle Dateien mit den vorgegebenen Endungen anwendbar ist, und auch automatisch fuer diese angewendet wird.

3.4 Variablen/Makros

Oft ist es bequemer, wenn man fuer lange oder haeufig verwendete Bezeichnungen eine abkuerzende Schreibweise einfuehren kann. Dies ist auch in make, mittels sog. Variablen oder Makros, moeglich. Diese Variablen stehen stellvertretend fuer den ihnen zugewießenen String. Die Variable wird von make vor dem Compilieren einfach per Textersetzung durch den ihr zugewiesenen String ersetzt.

Um eine Variable in make zu definieren, wird ihr einfach per Gleichheitszeichen ein String zugewießen, fuer den sie stellvertretend stehen soll. Zu beachten ist hierbei, dass gewisse Sonderzeichen nicht in einer Variablen enthalten sein duerfen. Diese sind ':', '=', '#' und das

Leerzeichen.

Um eine vorher definierte Variable in einem makefile zu verwenden, muß man sie in eine Art Huelle, bestehend aus einem '$' und einfachen Klammern, einfassen. Diese Huelle zeigt dem makefile, dass es sich bei dem in Klammern stehenden String um eine Variable handelt. Auf die Variable CC (Aufruf des C-Compilers) wird in einem makefile mittels $(CC) zugegriffen.

Zu beachten ist, dass viele Variablen in make bereits vordefiniert sind, und nicht extra definiert werden muessen, sondern direkt verwendet werden koennen.

3.5 Anweisungen

Eine Anweisung ist ein Sonderfall in einem makefile. Sie weißt das makefile an etwas besonderes zu tun, wie z.B. ein anderes makefile zu integrieren, oder einen Teil des makefiles ggf. zu ignorieren.

3.6 Kommentare

Wenn man einen Kommentar in ein makefile schreiben will, so schreibt man einfach ein '#' an den Beginn des Kommentars. Make betrachtet nun den Rest der Zeile als einen Kommentar, und ignoriert diese in der Ausfuehrung des makefiles.

3.7 “make clean”

Da beim Compilieren viele Dateien entstehen koennen, die man nach dem Compilieren gar nicht benoetigt, biete make eine Funktion, um sich dieser unerwuenschten Daten zu entledigen. Diese Funktion heißt “clean”.

Die Funktion “clean” wird definiert, indem man ein spezielles Tripel erstellt. Hierfuer schreibt man als Ziel einfach “clean” ohne irgend welche Abhaengigkeiten, und als Befehl einfach den Befehl, welcher alle ueberfluessigen oder unerwuenschten Dateien loescht. (z.B. unter Linux der rm-Befehl)

4. Fehlermeldungen

In einem makefile kann man eine ganze Reihe an Fehlern erzeugen. Die am haeufigsten begangenen Fehler sind:

- 'missing separator. Stop.' : Es fehlt ein <TAB> oder eine andere Trennung. (':', '=')
- 'No rule to make target `xxx'.': Die Regel, um ein Ziel zu erstellen, fehlt.
- 'warning: overriding commands for target `xxx`': Es sind mehrere Regeln fuer ein Ziel vorhanden, wobei die letzte gefundene Regel fuer das Ziel angewendet wird.
- `missing target pattern. Stop.': Es ist kein Makro fuer eine vorhandene Vorgabe vorhanden.

5. Anhang

5.1 Resumee

Das Tool make ist, gerade bei großen Projekten, geradezu unverzichtbar. Es ist ein sehr vielseitig anwendbares und aeußerst maechtiges Werkzeug. Da es sehr klar und einfach strukturiert ist, ist es, auch ohne großartige Einarbeitungszeit, sehr einfach und sicher zu verwenden.

5.2 Quellenangabe

- http://de.wikipedia.org/
- http://www.gnu.org/software/make/
- http://www.linuxfibel.de/