Einführung in make

Johannes Franken
<jfranken@jfranken.de>

make ist ein Interpretierer für Makefiles. Ein Makefile entspricht einem Shellscript mit Regieanweisungen, die **make** dazu befähigen, nur die gerade *erforderlichen* Zeilen auszuführen. Das spart Zeit.

Unglücklicherweise bezieht sich nahezu sämtliche verbreitete Dokumentation nur auf das Compilieren von C-Programmen. Diese Anleitung demonstriert, wie man *Alltags-Aufgaben* mit **make** beschleunigt, die insb. nicht mit der Programmiersprache C zusammenhängen.

Inhalt

Aufbau eines Makefile

Ein Makefile enthält im wesentlichen

- Definitionen von Variablen und Funktionen,
  z.B. **`myshell=bash`**
- Kommentare,
  z.B. **`# converting postscript to pdf`**
- Includes,
  z.B. **`-include Makefile.local`**
- Regeln,
  z.B. **`%.pdf: %.ps; -@ps2pdf $<`**

Explizite Regeln

Grundlagen

Eine Regel ist wie folgt aufgebaut:

```
Target [ weitere Targets] :[:] [ Vorbedingungen] [; Kommandos]
[  <tab>  Kommandos ]
[  <tab>  Kommandos ]
      ...
```

Die in eckigen Klammern dargestellten Teile sind optional.

Regeln *mit* Kommandos nennt man *explizit*, Regeln *ohne* Kommandos *implizit*. Dieses Kapitel widmet sich den *expliziten* Regeln.

Wichtig ist das Tabulatorzeichen (**<tab>**) vor den Kommandos ab der zweiten Zeile.

Mehrere Kommandos in einer Zeile kann man mit Semikolon trennen; überlange Zeilen sollte man aufteilen und mit Backslash verbinden.

Beispiel

Folgendes Makefile:

```
hello:
        @echo hello \
        world

diskfree:; df -h /
```

definiert zwei Regeln (Targets **hello** und **diskfree**) und ordnet ihnen (explizit...) jeweils ein kurzes Kommando zu. Wir können diese Kommandos gezielt aufrufen:

```
$ make hello
hello world
$ make diskfree
df -h /
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda6              43G   37G  3.6G  92% /
```

Der Klammeraffe (@) vor dem **echo** verhindert übrigens die Ausgabe des auszuführenden Befehls.

Feststellung 1: Wenn man beim **make**-Aufruf gar kein Target angibt, führt **make** einfach die erste Regel aus:

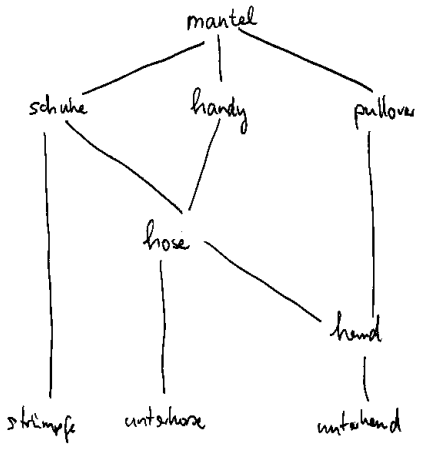
```
$ make
hello world
```

Feststellung 2: Wenn man beim **make**-Aufruf mehrere Targets angibt, führt **make** diese alle in der Reihenfolge aus:

```
$ make diskfree hello
df -h /
Filesystem            Size  Used Avail Use% Mounted on
/dev/hda6              43G   37G  3.6G  92% /
hello world
```

Abhängigkeiten

Die Installationsreihenfolge beim morgendlichen Anziehen ergibt sich aus folgendem Abhängigkeitsbaum:



Vor dem Anziehen der Schuhe sollten beispielsweise Strümpfe und Hose bereits angezogen sein. Wir sagen:

Die Schuhe erfordern Strümpfe und Hose.

und notieren im Makefile:

```
schuhe: struempfe hose
        @echo schuhe anziehen
```

Den Target-Namen erhält das Kommando über **$@**, damit können wir verallgemeinern zu:

```
schuhe: struempfe hose
        @echo $@ anziehen
```

Setzt man dieses Verfahren unter Verwendung der einzeiligen Notation für den gesamten Baum fort, so entsteht folgendes Makefile:

```
.PHONY: mantel schuhe handy pullover struempfe\
        hose hemd unterhose unterhemd

# Target    Vorbedingung           Befehl
# ------------------------------------------------
mantel:     schuhe handy pullover; @echo $@ anziehen
schuhe:     struempfe hose;        @echo $@ anziehen
handy:      hose;                  @echo $@ anziehen
pullover:   hemd;                  @echo $@ anziehen
struempfe:  ;                      @echo $@ anziehen
hose:       unterhose hemd;        @echo $@ anziehen
hemd:       unterhemd;             @echo $@ anziehen
unterhose:  ;                      @echo $@ anziehen
unterhemd:  ;                      @echo $@ anziehen
```

Die **.PHONY**-Anweisung bewirkt übrigens, dass die Targets nichts mit möglicherweise existierenden Dateien gleichen Namens zu tun haben. Mehr dazu im nächsten Beispiel.

Das Ergebnis stimmt optimistisch:

```
$ make mantel | nl
     1  struempfe anziehen
     2  unterhose anziehen
     3  unterhemd anziehen
     4  hemd anziehen
     5  hose anziehen
     6  schuhe anziehen
     7  handy anziehen
     8  pullover anziehen
     9  mantel anziehen
```

Angenommen, ich wollte nur den Pullover anziehen und alles, was dazu erforderlich ist:

```
$ make pullover | nl
     1  unterhemd anziehen
     2  hemd anziehen
     3  pullover anziehen
```

Das **|nl** gehört übrigens nicht zu **make**. Dieses Unix-Programm nummeriert nur die von **make** ausgegebenen Zeilen.

Bezug auf vorhandene Dateien

Aufgabe: Im vorigen Beispiel ging **make** davon aus, dass zu Beginn noch keine einzige Abhängigkeit erfüllt, man also vollständig nackt ist. Nach dem Anlegen der Grundausstattung trinke ich lieber erst mal einen Kaffee, bevor ich mir den Mantel usw. umwerfe. Benötigt wird eine Lösung für

```
$ make hemd hose struempfe
$ trinke kaffee
$ make mantel
```

Lösungsansatz: Das Berechnen der Schritte bis zum Mantel muss also auf Etappen verteilt und dazu der Zustand nach dem ersten **make** zwischengespeichert werden. Hierzu lege ich nach Abarbeitung eines Targets jeweils eine Datei an, die genauso heisst wie das Target. Die **.PHONY** Zeile kommt weg, dann geht **make** beim Vorhandensein der Dateien davon aus, dass das Target bereits erfüllt ist (genaugenommen vergleicht **make** das Dateidatum, mehr dazu im nächsten Kapitel).

Lösung: Das folgende Makefile:

```
# Target    Vorbedingung           Befehl
# ----------------------------------------------------------
mantel:     schuhe handy pullover; @echo $@ anziehen; touch $@
schuhe:     struempfe hose;        @echo $@ anziehen; touch $@
handy:      hose;                  @echo $@ anziehen; touch $@
pullover:   hemd;                  @echo $@ anziehen; touch $@
struempfe:  ;                      @echo $@ anziehen; touch $@
hose:       unterhose hemd;        @echo $@ anziehen; touch $@
hemd:       unterhemd;             @echo $@ anziehen; touch $@
unterhose:  ;                      @echo $@ anziehen; touch $@
unterhemd:  ;                      @echo $@ anziehen; touch $@
```

löst die Aufgabe wie folgt:

```
$ make hemd hose struempfe | nl
     1  unterhemd anziehen
     2  hemd anziehen
     3  unterhose anziehen
     4  hose anziehen
     5  struempfe anziehen
$ ls
hose       unterhemd   hemd   struempfe  unterhose
$ # trinke kaffee
$ make mantel | nl
     1  schuhe anziehen
     2  handy anziehen
     3  pullover anziehen
     4  mantel anziehen
$ ls
hemd    mantel    schuhe     unterhemd   handy
hose    pullover  struempfe  unterhose
```

Implizite Regeln

Ich kann jede dieser Regeln in zwei Teile teilen:

- Eine *implizite* Regel erklärt die Abhängigkeiten und
- eine *explizite* Regel die Kommandos.

Alle Regeln im letzten Beispiel führten die selben Kommandos aus und unterschieden sich nur in den Vorbedingungen. Für die Regeln ohne Vorbedingungen (z.B. **`struempfe`**) benötige ich keine implizite Regel. Wegen der identischen Kommandos kann ich die expliziten Regeln zusammenfassen. So entsteht folgendes, schon viel übersichtlichere Makefile:

```
# Eine explizite Regel definiert die Kommandos
mantel schuhe handy pullover struempfe hose\
hemd unterhose unterhemdr: ;  @echo $@ anziehen; touch $@

# Implizite Regeln klären die Vorbedingungen
mantel:     schuhe handy pullover
schuhe:     struempfe hose
handy:      hose
pullover:   hemd
hose:       unterhose hemd
hemd:       unterhemd
```

Eigene Variablen und Funktionen

Variablen werden mit **`=`** oder **`:=`** deklariert, je nachdem ob die ggf. enthaltenen weiteren Variablen und Funktionen bei *Verwendung* oder *Definition* der Variable ausgewertet werden sollen. Den in einer Variable gespeicherten Text erhält man mit **`$(myvar)`**. Mit **`$(call myvar)`** führt **`make`** diesen (wie eine Funktion) aus.

```
# Definition einer Variable
kleidungsstuecke= mantel schuhe handy pullover\
        struempfe hose hemd unterhose unterhemd

# Kommando für alle Targets
$(kleidungsstuecke):; @echo $@ anziehen; touch $@

# Vorbedigungen für einige Kleidungsstücke
mantel:     schuhe handy pullover
schuhe:     struempfe hose
handy:      hose
pullover:   hemd
hose:       unterhose hemd
hemd:       unterhemd

# Zusaetzlicher Funktionsumfang
.PHONY: nackt
nackt:      ; @-rm $(kleidungsstuecke)
```

Das Minuszeichen (**`-`**) vor dem **`rm`** bewirkt übrigens, dass Fehler beim **`rm`** nicht zum Abbruch von **`make`** führen, also **`make nackt mantel`** auch dann im Mantel endet, wenn man vorher bereits nackt war.

Mehrere Makefiles verbinden

Teile und herrsche! Größere Projekte (z.B. der Linux-Kernel) bringen oft so gewaltige Makefiles mit sich, dass man erst durch Aufteilen auf mehrere, kleinere Dateien den Überblick gewinnt.

Mit dem **`include`**-Befehl kann man weitere Makefiles an Ort und Stelle einfügen. Wenn man ihm ein Minuszeichen voranstellt, bricht **`make`** beim Fehlen dieses Files nicht ab.

Beispiel:

```
# Krawatte anziehen
-include Makefile.krawatte
```

: :-Regeln

Das folgende Makefile definiert das Target **`struempfe`** verbotenerweise mehrfach:

```
struempfe: ; @echo linken Strumpf anziehen
struempfe: ; @echo rechten Strumpf anziehen
```

`make` gibt daher eine Warnmeldung aus und führt nur das Kommando der letzten Deklaration aus:

```
$ make
Makefile:2: warning: overriding commands for target ‘struempfe’
Makefile:1: warning: ignoring old commands for target ‘struempfe’
rechten Strumpf anziehen
```

Wenn man möchte (und das ist insb. bei der Arbeit mit includes oft der Fall), dass die Kommandos *beider* Deklarationen ausgeführt werden, verwende man doppelte Doppelpunkte:

```
struempfe:: ; @echo linken Strumpf anziehen
struempfe:: ; @echo rechten Strumpf anziehen
```

Und siehe da:

```
$ make
linken Strumpf anziehen
rechten Strumpf anziehen
```

Dateidatum

Folgendes Makefile hilft mir bei der Konvertierung einer Postscript- Datei (**`tiger.ps`**) ins PDF-Format (**`tiger.pdf`**):

```
tiger.pdf: tiger.ps; ps2pdf $<
```

`make` wird den **`ps2pdf`** nur dann aufrufen, wenn **`tiger.pdf`** nicht existert oder *älter* ist als **`tiger.ps`**. Die Variable **`$<`** wird übrigens automatisch ersetzt durch den Inhalt der Vorbedingung, also **`tiger.ps`**.

So sieht das Ergebnis aus:

```
$ ls tig*
tiger.ps
$ make tiger.pdf
ps2pdf tiger.ps
$ ls tig*
tiger.pdf tiger.ps
$ make tiger.pdf
make: ‘tiger.pdf’ is up to date.
$
$ cp tiger2.ps tiger.ps
$ make tiger.pdf
ps2pdf tiger.ps
```

Patterns

Dank der *Pattern*-Schreibweise muss ich nicht für jede Postscript-Datei eine eigene Regel anlegen. Ich verallgemeinere das vorangegangene Beispiel zu:

```
%.pdf: %.ps; -ps2pdf $<
```

Das **`make tiger.pdf`** funktioniert damit unverändert. Allerdings funktioniert der Aufruf von **`make`** (ohne Parameter) dann nicht mehr; man muss explizit ein Target übergeben.

```
$ make
make: *** No targets.  Stop.
$ make tiger.pdf
ps2pdf tiger.ps
```

Folgendes Makefile erstellt die Datei **`tiger.pdf`** auch wenn man **`make`** ohne Parameter aufruft:

```
all  : tiger.pdf
%.pdf: %.ps; -ps2pdf $<
```

Wenn ich möchte, dass beim Aufruf von **`make`** gleich *alle* im Verzeichnis enthaltenen Postscriptdateien konvertiert werden, so kann mir die **`$(wildcard)`**-Funktion eine Liste aller Postscriptdateien liefern, die ich mit der **`$(patsubst)`**-Funktion in eine Liste der PDF-Dateien umwandle:

```
all: $(patsubst %.ps,%.pdf,$(wildcard *.ps))
%.pdf: %.ps; -ps2pdf $<
```

Eingebaute Funktionen

$(subst from,to,text)	Ersetze **from** durch **to** im **text**.
$(patsubst pattern,replacement,text)	Ersetze im **text** alle Wörter, die **pattern** enthalten, durch **replacement**.
$(strip string)	Entferne alle überflüssigen Whitespaces aus **string**.
$(findstring find,text)	Suche nach **find** im **text**.
$(filter pattern...,text)	Lösche alle Wörter aus **text**, auf die keines der **pattern** zutrifft.
$(filter-out pattern...,text)	Lösche alle Wörter aus **text**, auf die eines der **pattern** zutrifft.
$(sort list)	Gibt die **list** in sortierter Reihenfolge zurück, ohne Doppelte.
$(dir names...)	Gibt das Verzeichnis aller übergebenen **names** zurück.
$(notdir names...)	Gibt den reinen Dateinamen aller übergebenen **names** zurück.
$(suffix names...)	Gibt die Dateiendung (ab dem letzen Punkt) aller übergebenen **names** zurück.
$(basename names...)	Gibt den Stamm (Dateiname ohne Endung) aller übergebenen **names** zurück.
$(addsuffix suffix,names...)	Hänge **suffix** an alle übergebenen **names** an.
$(addprefix prefix,names...)	Hänge den **prefix** vor alle übergebenen **names**.
$(join list1,list2)	Verbinde die beiden Wortlisten
$(word n,text)	Liefert das **n**-te Wort aus dem **text**.
$(words text)	Gibt die Zahl der Wörter in **text** zurück.
$(wordlist s,e,text)	Liefert den Text vom **s**-ten bis zum **e**-ten Wort.
$(firstword names...)	Liefert das erste Wort von **names**.
$(wildcard pattern...)	Gibt eine Liste aller Dateien zurück, die eine gewöhnliche Shell auf das **pattern** matchen würde.
$(error text...)	Erzeugt einen fatalen Fehler mit Hinweis **text**. Dabei beendet sich **make**.
$(warning text...)	Zeigt eine Warnung mit Hinweis **text** an.
$(shell command)	Fürt den **command** in der Shell aus und gibt seine Ausgabe zurück.
$(origin variable)	Liefert einen Text zurück, aus dem hervorgeht, wie die Variable **variable** definiert worden war.
$(foreach var,words,text)	Evaluate **text** with **var** bound to each word in **words**, and concatenate the results.
$(call var,param,...)	Führe den Inhalt der Variablen **var** wie eine Funktion aus. Innerhalb der Funktion kann man mit **$(1),$(2)** auf die Parameter zugreifen.

Automatische Variablen

$@	Der Name des Target.
$%	Der Member-Name, falls das Target sich in einem Archiv befindet.
$<	Der Name der ersten (oder einzigen) Vorbedingung.
$?	Leerzeichen-getrennte Liste all derer Dateien aus der Vorbedingung, die neuer als das Target sind.
$^** **$+	Leerzeichen-getrennte Liste aller Vorbedingungen. Bei **$^** sind zusätzlich die Doppelten entfernt.
$*	Der Stamm des Dateinamens, auf den die implizite Regel zugetroffen hatte.
$(@D) **$(@F)**	Verzeichnis und Dateiname von **$@**
$(*D) **$(*F)**	Verzeichnis und Dateiname von **$***
$(%D) **$(%F)**	Verzeichnis und Dateiname von **$%**
$(<D) **$(<F)**	Verzeichnis und Dateiname von **$<**
$(^D)** **$(^F)	Verzeichnis und Dateiname von **$^**
$(+D) **$(+F)**	Verzeichnis und Dateiname von **$+**
$(?D) **$(?F)**	Verzeichnis und Dateiname von **$?**

Verweise

- GNU **make** Manual