SelfLinux-0.12.3

Make

Autor: Oliver Böhm (*boehm@2xp.de*)
Formatierung: Florian Frank (*florian@pingos.org*)

Inhaltsverzeichnis

1 Einführung

Stellen Sie sich vor, Sie möchten ein kleines Programm übersetzen und müssen dazu zu Ihrem Rechner lediglich ein **mach mal** sagen und Ihr Rechner würde Sie verstehen. Genau zu diesem Zweck gibt es das `make`-Kommando. Doch ein

```
user@linux / $ make love
```

wird vom Rechner leider mit einem

```
don't know how to make 'love'
```

quittiert. (das **GNU-make** gibt leider keine so schöne Fehlermeldung aus). Im Verlauf dieses Kapitels werden wir etwas **Aufklärungsarbeit** für unser Unix-System betreiben, damit es unseren Wunsch erfüllt.

2 Grundlagen

2.1 Ein einfaches Makefile

Basis für unsere Aufklärungsbemühungen ist ein **Makefile**, aus dem das `make`-Kommando die nötigen Informationen bezieht, um **love** zu machen:

```
#
#   Makefile for making love
#

love : love.c
        gcc love.c -o love
```

Jetzt weiss das `make`-Kommando, dass es nach einem **love.c** Ausschau halten muss, um dann mit Hilfe des Kommandos **gcc love.c -o love** (Aufruf des GNU-C-Compilers) ein **love** zu generieren.

Wenn Sie das Ganze ausprobieren wollen, achten Sie bitte darauf, dass vor **gcc ...** ein **Tabulator-Zeichen** steht. Für **love.c** können Sie folgende C-Quelle verwenden:

love.c

```
main()
{
    printf("Hallo Schatz!\n");
    return 0;
}
```

Jetzt haben wir alle Hilfsmittel beisammen, um den Rechner zu einem `make love` aufzufordern:

```
user@linux / $ make love

gcc love.c -o love
```

Und tatsächlich: der Rechner erwidert unseren Wunsch mit dem Aufruf des GNU-Compilers **gcc...**. Sollten Sie eine Fehlermeldung (zum Beispiel **Makefile:6: *** missing separator...**) erhalten, überprüfen Sie bitte nochmal das Makefile: die Zeile mit **gcc...** muss mit einem **Tabulator-Zeichen** beginnen! Wenn Sie das Verzeichnis auflisten (zum Beispiel mit `ls -l`), werden Sie feststellen, dass das `make`-Kommando tatsächlich ein **love** erzeugt hat, das sich aufrufen lässt:

```
user@linux / $ ./love

Hallo Schatz!
```

Die beiden Schritte, Kompilation und Programmaufruf, lassen sich natürlich auch zusammen im Makefile unterbringen:

```
love : love.c
	gcc love.c -o love
	./love
```

Damit wird nach erfolgreicher Kompilation das `love`-Programm aufgerufen.

2.2 Regel-Werk

Ein Makefile enthält Regeln, die die Abhängigkeiten der Quellen zum Programm, zu Objekt-Dateien, zu Bibliotheken oder auch anderen Dateien definiert. Eine Regel hat dabei typischerweise folgendes Aussehen:

```
target: dependencies
	actions
```

Bei **target** handelt es sich um das Ziel, das gebaut werden soll (im obigen Beispiel war es **love**). Die **dependencies** geben an, wovon das **target** abhängt (im Beispiel hängt `love` von love.c ab). Hinter **actions** verbirgt sich das (oder die) Kommando(s), das/die zum Bau des Targets notwendig ist/sind. Achten Sie bitte darauf, dass die Zeile mit den **actions** mit einem **Tabulator-Zeichen** beginnen muss. **actions** darf dabei aus mehreren Zeilen (= mehrere Kommandos) bestehen, aber alle diese Zeilen müssen mit dem **Tabulator-Zeichen** beginnen.

Achtung! Leider unterscheidet sich ein Tabulator-Zeichen optisch nicht von 8 Leerzeichen. Das `make`-Kommando sieht das leider etwas enger und gibt stattdessen eine wenig hilfreiche Fehlermeldung (wie **...missing separator...** oder Ähnliches) aus.

Wenn Sie Kommentare in einem Makefile verwenden wollen, leiten Sie diese mit dem #-Zeichen ein. Ein Kommentar erstreckt sich von # bis Zeilenende.

2.3 Abhängigkeiten

Wie kann `make` wissen, wann es ein Ziel zu bauen hat? Wenn Sie erneut ein `make love` versuchen, werden Sie von `make` einen Korb bekommen:

```
user@linux / $ make love

make: 'love' is up to date.
```

Das Geheimnis liegt in den Abhängigkeiten: immer, wenn das Ziel (**love**) jünger als seine Abhängigkeiten (love.c) ist, geht `make` davon aus, dass es nichts zu tun gibt. Wenn Sie jetzt love.c editieren und abspeichern, ändert sich dies: jetzt ist love.c jünger. Ein `make love` wird daraufhin wieder den C-Compiler aufrufen.

Sie werden sich vielleicht fragen: "Wozu der ganze Aufwand? Ich kann doch den C-Compiler manuell aufrufen, wenn ich "love.c" editiere". Aber bei größeren Programmen hat man nicht nur eine C-Quelle, sondern mehrere. Hier hilft das `make`-Kommando, den Aufruf des Compilers zu automatisieren.

```
#
#   Makefile mit mehreren Abhängigkeiten
#

children : mummy.o daddy.o
        gcc mummy.o daddy.o -o children

mummy.o : mummy.c
        gcc -c mummy.c

daddy.o : daddy.c
        gcc -c daddy.c
```

Wie man an diesem Beispiel sieht, können Abhängigkeiten auch mehrstufig sein. **children** hängt von **mummy.o** und **daddy.o** ab. **mummy.o** und **daddy.o** sind Objekt-Dateien, die wiederum von **mummy.c** und **daddy.c** abhängen. Wenn Sie das erste Mal ein `make children` eingeben, werden zuerst **mummy.o** und **daddy.o** gebaut, ehe daraus dann **children** erzeugt wird:

```
user@linux / $ make children

gcc -c mummy.c
gcc -c daddy.c
gcc mummy.o daddy.o -o children
```

Wenn Sie jetzt eine der beiden C-Quellen verändern und erneut das `make`-Kommando aufrufen, wird nur die Objekt-Datei neu erzeugt, die **veraltet** ist.

2.4 Pseudo-Ziele

Dies sind Ziele, die keine Abhängigkeiten besitzen. Damit können Sie aber nie **up-to-date** werden mit dem gewünschten Nebeneffekt, dass von make immer die Aktionen ausgeführt werden, wenn solch ein Pseudo-Ziel aufgerufen wird:

```
clean :
	rm *.o core
```

Jedes Mal, wenn der Benutzer ein `make clean` eingibt, wird vom make-Kommando ein `rm *.o core` aufgerufen, d.h. es werden sämtliche Objekt-Dateien (*.o) und core-Dateien (diese werden bei einem Programm-Absturz angelegt) gelöscht.

Weil das Ziel **clean** nie erzeugt wird, wird es als Pseudo-Ziel bezeichnet.

3 Makros

3.1 Syntax

Man kann auch Variablen innerhalb von Makefiles verwenden. Sie werden dort als Makros bezeichnet und meistens gross geschrieben. Mit

```
CC = gcc
```

wird ein Makro **CC** mit dem Wert `gcc` angelegt. Der Wert darf dabei auch aus mehreren Wörtern bestehen (Beispiel: CC = `gcc -O`). Die Definition darf sich auch über mehrere Zeilen erstrecken, wenn die Zeile mit \ aufhört (**Achtung**: bitte darauf achten, dass das \ tatsächlich das letzte Zeichen in der Zeile ist und kein Tabulator- oder Leerzeichen dahintersteht!).

Der Aufruf des Makros **CC** kann mit

```
$(CC)
${CC}
```

erfolgen.

3.2 Vordefinierte Makros

`make` kennt schon eine Reihe von vordefinierten Makros. So ist zum Beispiel die Makrodefinition **CC = cc** dem `make`-Kommando bereits bekannt. Mit

```
user@linux / $ make -p -f /dev/null
```

können die internen Makros und Regeln abgefragt werden. Ist man nur an den Makros interessiert, kann man diese mit Hilfe des `grep`- und `sort`-Kommandos ausfiltern und sortieren:

```
user@linux / $ make -p -f /dev/null | grep " = " | sort
```

3.3 Makro-Übergabe

Makros können schon vordefiniert sein, Makros können im Makefile gesetzt werden, Makros können aber auch beim Aufruf von `make` übergeben werden:

```
user@linux / $ make love CC=gcc
```

Soll das Makro aus mehreren Wörtern bestehen, so muss es beim Aufruf gequotet (d.h. mit einfachen oder doppelten Anführungszeichen umgeben) werden:

```
user@linux / $ make love CFLAGS="-O -DNDEBUG"
```

3.4 Umgebungs-Variablen

Umgebungs-Variablen werden innerhalb eines Makefiles genauso wie ein Makro angesprochen. Damit kann man zum Beispiel eine Variable **CXX** setzen, die den GNU-C++-Compiler enhalten soll:

Bash:

```
user@linux / $ export CXX=g++
```

C-Shell:

```
user@linux / $ setenv CXX g++
```

Die Umgebungs-Variable **CXX** steht damit im Makefile als Makro zur Verfügung, was in der folgenden Regel ausgenutzt wird:

```
hello : hello.cpp
        $(CXX) hello.cpp -o hello
```

3.5 Prioritäts-Regeln

Wenn man dasselbe Makro auf verschiedene Weisen definieren kann, ist es wichtig, zu wissen, welches Makro im Falle eines Konfliktes gewinnt. Makros werden nach folgender Reihenfolge aufgelöst:

1. vordefinierte Makros
2. Umgebungs-Variablen
3. interne Makros
4. Makros über die Kommandozeile

Die Priorität ist aufsteigend, das heisst Makros, die über die Kommandozeile gesetzt werden, gewinnen immer.

3.6 String-Substitution

Makros dürfen sich auf andere Makros beziehen. Betrachten wir dazu folgendes Beispiel:

```
C_FILES   = daddy.c mummy.c
H_FILES   = children.h
SRC_FILES = $(C_FILES) $(H_FILES)
OBJ_FILES = daddy.o mummy.o
```

Das Mako **SRC_FILES** bezieht sich dabei auf die beiden Makros **C_FILES** und **H_FILES**. Vergleichen wir jetzt das Makro **C_FILES** mit **OBJ_FILES**, so stellen wir fest, dass sie bis auf die Endung (.c bzw. .o) identisch sind. Damit wir nicht jedes Mal die OBJ_FILES anpassen müssen, wenn sich C_FILES ändert, gibt es die String-Substitution:

```
OBJ_FILES = $(C_FILES:.c=.o)
```

Damit hängt OBJ_FILES direkt von C_FILES ab, das heißt wenn in C_FILES eine neue .c-Datei aufgenommen wird, müssen wir OBJ_FILES nicht ändern.

3.7 Interne Makros

Neben den vordefinierten Makros gibt es eine Reihe von internen Makros, die manche Regel vereinfachen können:

$@	Name des aktuellen Zieles
$?	Name der abhängigen Dateien, die neuer als das Ziel sind
$<	Name der abhängigen Dateien, die neuer als das Ziel sind. (mit Endung)
$*	Name der abhängigen Dateien, die neuer als das Ziel sind. (ohne Endung)
$%	Name der entsprechenden Objekt-Datei (.o), falls das Ziel eine Bibliothek ist.

Beispiel:

```
children: $(OBJ_FILES)
        $(CC) $(CFLAGS) $(OBJ_FILES) -o $@
```

In diesem Beispiel wird von make **$@** durch den Namen des Ziels (**children**) ersetzt.

Will man auf den Dateinamen oder Verzeichnisnamen zugreifen, kann man dies (ausser bei **$?**) über die Modifier **F** (wie File) oder **D** (wie Directory).

Beispiel:

```
$(*D), $(@F)
```

4 Suffix-Regeln

Wenn Sie eine Datei mit der Endung **.c** sehen, wissen Sie, dass es sich um eine C-Datei handelt. Vermutlich wissen Sie auch, dass Sie einen C-Compiler aufrufen müssen, um daraus eine Objekt-Datei mit der Endung **.o** zu generieren.

Über die Suffix-Regeln kann man make dieses Wissen, wie es aus einer **.c**-Datei eine **.o**-Datei machen soll, beibringen:

```
#
#   Suffix-Regeln fuer C-Programme
#

.SUFFIXES : .c .o .i

.c.o :
        $(CC) -c $<

.c.i :
        $(CC) -E $< > $@
```

Mit der Zeile **.SUFFIXES** gibt man dem `make`-Kommando bekannt, welche Endungen es kennen soll. Sollen außer den drei angegebenen Endungen **.c**, **.o** und **.i** die bereits bekannten Endungen weiter gelten, kann man **$(SUFFIXES)** noch auf der rechten Seite ergänzen.

Nach der Bekanntgabe der Endungen folgen die Suffix-Regeln: Die Regel **.c.o** gibt an, wie make aus einer .c-Datei eine .o-Datei machen soll, und die Regel **.c.i** besagt, wie das Ergebnis des Präprozessors in einer Datei mit der Endung .i landet.

Beispiel:

```
user@linux / $ make love.i

gcc -E love.c > love.i
```

4.1 Null-Suffix-Regeln

Null-Suffix-Regeln sind daran erkennbar, dass die zweite Endung fehlt:

```
#   Null-Suffix-Regel

.c:
        $(CC) $< -o $@
```

Null-Suffix-Regeln kommen dann zum Einsatz, wenn direkt aus einer Quell-Datei ein ausführbares Programm ohne Datei-Endung erzeugt werden soll.

Beispiel:

```
user@linux / $ make love
gcc love.c -o love
```

4.2 Eingebaute Suffix-Regeln

Viele Regeln hat make schon eingebaut, nicht nur für C und C++. Die Regeln und Makros können über

```
user@linux / $ make -p -f /dev/null
```

ausgegeben werden. Wie man dabei sieht, gibt es auch für andere Sprachen bereits vordefinierte Regeln.

Endung	Bedeutung
.c	C-Source
.cc, .C, .cpp	C++-Source
.f	Fortran
.gz	komprimierte Datei (gzip)
.h	Header-Datei (C, C++)
.i	Dateien nach dem Präprozessor-Lauf
.l	lex-Dateien
.o	Objekt-Dateien
.p	Pascal-Dateien
.s	Assembler-Dateien
.y	yacc-Dateien
.Z	komprimierte Dateien (compress)

4.3 Konflikt-Behandlung

Betrachten wir ein Makefile, das nur aus folgender Regel besteht:

```
children : mummy.o daddy.o
```

Dummerweise sind mehrere Suffix-Regeln definiert, wie eine .o-Datei erzeugt werden kann, zum Beispiel aus einer .c-Datei oder aus einer .cpp-Datei oder Datei mit anderer Endung. Wenn nur eine entsprechende .c-Datei da ist, ist alles klar. Aber was, wenn nicht? Dann wird die Such-Reihenfolge durch die SUFFIXES-Liste bestimmt:

```
.SUFFIXES: .o .c .cpp .f
```

Es wird zuerst nach einer .c-Regel gesucht (.o nach .o macht keinen Sinn), dann nach einer .cc-Regel und so fort.

4.4 Eigene Suffix-Regeln

Eigene Suffix-Regeln wird man dann einführen, wenn

* die eingebauten Regeln nicht ausreichen,
* man mit den eingebauten Regeln nicht einverstanden ist.

5 Kommandos

Da die Kommandos in einer eigenen (Unter-)Shell ausgeführt werden, stehen alle Möglichkeiten einer Shell wie Wildcards, Ein-/Ausgabe-Umlenkung, ... zur Verfügung.

5.1 Wildcards

Wildcards (im Deutschen auch als Jokerzeichen bezeichnet) können innerhalb von Kommandos verwendet werden, aber auch in den Abhängigkeiten einer Regel auftauchen:

```
print : *.c
        lpr *.c
```

Leider haben diese Wildcards eine leicht unterschiedliche Bedeutung: das erste Wildcard ***.c** innerhalb der Abhängigkeit wird von `make` aufgelöst, das Wildcard im Kommando (**lpr *.c**) von der Shell. Während bei der Shell die **Hidden Files** (das sind diejenige Dateien wie zum Beispiel .bashrc, die mit einem **.** anfangen) nicht dazugehören, ist hier das `make`-Kommando anderer Auffassung.

5.2 Mehrere Kommandos

Jede Zeile im Aktions-Teil wird in einer eigenen Unter-Shell gestartet. Will man mehrere Kommados auf einmal absetzen, werden diese hintereinander aufgereiht und jeweils durch einen Strichpunkt (**;**) voneinander getrennt:

```
clean :
        cd obj ; rm *.o
```

Bei diesem Beispiel ist zu bedenken, dass das zweite Kommando (`rm *.o`) auch dann ausgführt wird, wenn der erste Teil (`cd obj`) nicht geklappt hat. Deswegen können Kommandos auch über **&&** (UND) verknüpft werden:

```
clean :
        cd obj && rm *.o
```

Vorteil der UND-Verknüpfung: Schlägt das erste Kommando (`cd obj`) fehl, wird das zweite Kommando (`rm *.o`) auf keinen Fall ausgeführt.

Man kann die Kommandos auch auf mehrere Zeilen verteilen, indem man sie mit dem Backslash (\) als allerletztes Zeichen in der Zeile verbindet:

```
clean :
        cd obj &&       \
        rm *.o
```

Dabei ist darauf zu achten, dass der Backslash (\) wirklich das letzte Zeichen in der Zeile ist -- er hebt nämlich

die Bedeutung des nächsten Zeichens (in diesem Fall das Zeilenende) auf. Steht hier ein Leerzeichen, wird die (nicht vorhandene) Sonderbedeutung des Leerzeichens aufgehoben.

5.3 Skript-Programmierung

Für komplexere Aufgaben kann man eigene Skripte schreiben oder aber auch direkt im Makefile Programmierkonstrukte der (Bourne-)Shell verwenden (das `indent`-Kommando formartiert C-Quellen nach den GNU-C-Konventionen):

```
indent :
        for file in $(C_FILES) ; do     \
            indent $$file ;             \
        done
```

Hier ist darauf zu achten, dass die einzelnen Konstrukte durch einen Strichpunkt (**;**) voneinander getrennt werden. Erstrecken sich die Kommandos über mehrere Zeilen, ist dies durch ein Backslash (\) am Zeilenende zu kennzeichnen.

Der Zugriff auf Shell-interne Variablen wird durch ein zusätzliches Dollar-Zeichen (**$**) bewerkstelligt (siehe **$$file** im Beispiel).

5.4 Fehlercode

Wenn ein Kommando einen Fehlercode zurückliefert, wird make an dieser Stelle abgebrochen. Dabei ist unter Unix alles, was einen Rückgabewert ungleich 0 zurückliefert, ein Fehlercode.

Meistens ist dieses Verhalten ja gewünscht, zum Beispiel beim Kompilieren (was der Normalfall sein dürfte). Aber es gibt auch Situationen, in dem der Rückgabewert eines Kommandos egal ist, zum Beispiel in

```
clean :
        - rm core
        - rm *.o
```

ist es nicht weiter tragisch, wenn `rm core` nicht erfolgreich war (weil es nicht existiert). Der Abbruch im Fehlerfall wird durch ein vorangestelltes Minus (**-**) vermieden.

Manche Kommandos, wie zum Beispiel das `rm`-Kommando, können meist über die Option `-f` gezwungen werden, fehlende Dateien oder andere Ungereimtheiten zu ignorieren. Damit lässt sich das vorige Beispiel auch folgendermassen formulieren:

```
clean :
        rm -f core
        rm -f *.o
```

5.5 Ausgabe unterdrücken

Normalerweise wird jedes Kommando, das ausgeführt wird, vorher ausgegeben. Mit einem vorangestellten **@** kann man das unterdrücken:

```
statistic :
        @ echo ""
        @ echo "      S T A T I S T I C S"
        @ echo ""
        @ echo "   Lines   Words   Bytes Modul"
        @ echo "   -------------------------------"
        @ wc ${SRC_FILES}
        @ echo ""
```

Vor Allem beim `echo`-Kommando macht es keinen Sinn, dass vor der eigentlichen Ausgabe das `echo`-Kommando samt Ausgabe-String ausgegeben wird.

6 Projekt-Management

Für kleinere Programmpakete ist das Makefile meist noch recht übersichtlich. Allerdings verleiten die vielen Möglichkeiten, die make bietet, dazu dass man sich schnell im **Regel-Dschungel** die Orientierung verliert und das Makefile immer mehr ein undurchsichtiges Eigenleben entwickelt. Und wehe, das Wissen über den Aufbau und Ablauf der Makefiles konzentriert auf einen einzelnen Spezialisten. Wenn dieser dann nicht mehr zur Verfügung steht, kann die weitere Pflege und Wartung der Makefiles den eigentlichen Entwicklungsaufwand übersteigen.

Während man bei kleineren Projekten das Makefile notfalls nochmals aufsetzen kann, ist dies bei größeren Projekten oft mit erheblichem Aufwand verbunden. Daher sollte man auch (oder gerade) bei Makefiles nach dem Motto

> **So einfach wie möglich - aber nicht einfacher**

handeln.

Dieses Kapitel beschäftigt sich mit verschiedenen Aspekten, wie man mit `make` größere Projekte verwalten kann. Später werden wir Makefile-Richtlinien kennenlernen, die die Einarbeitung und Wartung von (eigenen oder fremden) Makefiles erleichtern können.

6.1 Schwierigkeiten

Einige der Probleme, die den Einsatz von `make` erschweren, sind:

* Verzeichnis-Baum
* Bedingte Kompilierung (#if ... #endif)
* versteckte Abhängigkeiten (zum Beispiel über Header-Dateien)
* Versionierung

Manche der Probleme resultieren aus Unzulänglichkeiten des Compilers, manche resultieren aus Annahmen und Beschränkungen einiger Unix-Werkzeuge. Diese spiegeln sich zum Teil in den eingebauten Suffix-Regeln wieder.

Ursprünglich war `make` nur zur Vereinfachung der Kompilierung gedacht, hat sich aber über die Jahre zu einem mächtigen Entwicklungswerkzeug gemausert. Nicht zuletzt auch deswegen, weil es inzwischen eine Reihe von Werkzeugen gibt, die um make herum gebaut wurden, um die Einschränkungen aufzuheben.

6.2 Dummy-Ziele

In der Praxis werden Dummy-Ziele recht häuig eingesetzt, um mehrere Ziele zusammenzufassen:

```
# compile all

all : anna berta carmen

anna : anna.c
        $(CC) -g -o anna anna.c

berta : berta.c
        $(CC) -g -o berta berta.c

carmen : carmen.c
```

```
	$(CC) -g -o carmen carmen.c
```

Der Entwickler braucht nur `make all` einzugeben und sämtliche Programme werden übersetzt.

6.3 Timestamp-Ziele

Eine etwas subtilere Variante von Dummy-Zielen sind **Timestamp**-Targets. Damit werden Ziele bezeichnet, die zwar angelegt werden, aber nicht als Ziel gebraucht werden. In Wirklichkeit werden sie zur Synchronisation von Aktivitäten verwendet:

```
TIMESTAMP.strip : anna berta carmen
	strip $?
	touch $@
```

Was macht dieses Ziel? Falls **TIMESTAMP.strip** nicht existiert oder eines der abhängigen Dateien **anna**, **berta** oder **carmen** neuer ist, werden die entsprechenden Dateien ge-`strip`-t (das `strip`-Kommando entfernt die Symbol-Tabelle aus dem Programm. Dadurch wird das Programm kürzer, kann aber dafür nicht mehr debuggt werden.) und danach **TIMESTAMP.strip** angelegt bzw. mit einem neuen Zeitstempel versehen (über das `touch`-Kommando).

Die Datei **TIMESTAMP.strip** dient also nur dazu, festzustellen ob **anna**, **berta** oder **carmen** schon einen `strip`" hinter sich haben. Diesen Trick findet man häufiger in Makefiles. Wenn Sie sich also schon immer gefragt haben, zu was Dateien der Größe **0** gut sein sollen, hier ist eine mögliche Antwort.

Allerdings hat diese Lösung auch einen Haken: man sieht der Datei **TIMESTAMP.strip** nicht an, zu was sie gut sein soll und ein ordnungsliebender Mensch könnte leicht auf die Idee kommen, diese Datei zu löschen, da sie die Größe **0** hat - weg damit! Daher ist es besser, dieser Datei einen sinnvollen Inhalt zu geben, damit sie

* eine Größe > 0 hat und
* damit der ahnungslose Benutzer einen Schimmer bekommt, zu was diese Datei gut sein könnte.

Dies kann man zum Beispiel durch folgende Regel erreichen:

```
TIMESTAMP.strip : anna berta carmen
	strip $?
	echo "last strip of anna, berta or carmen:" > $@
	date >> $@
```

6.4 Rekursives make

Am wenigsten problematisch ist es, wenn man sämtliche Dateien in einem einzigen Verzeichnis hat. Leider ist dieses Vorgehen bei größeren Projekten nicht praktikabel und üblicherweise hat man seine Dateien über mehrere Verzeichnisse verteilt.

6.4.1 Verteiltes make

Eine Möglichkeit, mit dem Verzeichnisbaum fertig zu werden, besteht darin, in jedes Verzeichnis ein Makefile

zu plazieren, das über das Makefile im übergeordneten Verzeichnis aufgerufen wird.

Das oberste Makefile könnte dabei folgendermaßen aussehen:

```
SUBDIRS = src lib

all :
        for d in $(SUBDIRS); do \
            (cd $$d; make all)  \
        done
```

Voraussetzung dafür ist natürlich, dass die drunterliegende Makefiles ein Ziel **all** besitzen.

6.4.2 Weitergabe von Makros

Der rekursive Aufruf von make ist auch dazu geeignet, Informationen und Flags durchzureichen.

Beispiel:

```
CFLAGS     = -O
DEBUGFLAGS = -g $(CFLAGS)

testbin :
        make bin "CFLAGS=$(DEBUGFLAGS)"
```

In diesem Beispiel wird durch `make testbin` dasselbe Makefile noch ein Mal aufgerufen, jedoch mit geänderten **CFLAGS**. Mit demselben Verfahren können auch Makros in drunterliegenden Makefiles überschrieben werden.

GNU-`make` und die meisten `make`-Versionen besitzen auch ein internes **MAKE-Makro**. Damit lautet die obere **testbin**-Regel:

```
testbin :
        $(MAKE) bin "CFLAGS=$(DEBUGFLAGS)"
```

Der Vorteil des internen **MAKE-Makros** ist die Weitergabe der Optionen beim Aufruf von `make`. Wird beispielsweise make mit der Option `-n` aufgerufen, so wird damit auch alle weiteren makes mit `-n` aufgerufen (die Option `-n` zeigt nur die Kommandos an, führt sie aber nicht aus).

6.4.3 Wichtige Makros

Jedes Makefile hat seine eigenen Makros. Um die Verwaltung und Verwirrung gering zu halten, sollte jedes Makefile dieselben Makro-Namen besitzen. Jeder `make`-Aufruf sollte wichtige Makros weiterreichen.

Beispiel:

```
all :
        for d in $(SUBDIRS); do       \
            (cd $$d;                  \
```

```
            make all "CFLAGS=$(CLAGS) \
                LDFLAGS=$(LDFLAGS)    \
                LIBFLAGS=$(LIBFLAGS)) \
        done
```

6.5 Header-Dateien

Objekt-Dateien hängen nicht nur von C-Sourcen, sondern auch von Header-Dateien ab, d.h. man müsste diese eigentlich mit in die Abhängigkeiten aufnehmen:

```
love.o : love.c darling.h
        $(CC) love.c
```

Dies wird man aber in den seltensten Fällen in Makefiles antreffen, und zwar meist aus folgenden Gründen:

* Faulheit des Programmierers
* versteckte Abhängigkeiten
* zu dynamisch
* zu großer Overhead

Glücklicherweise erhält der Programmierer hier Unterstützung vom (GNU-)Compiler: Mit der Option `-M` generiert der Compiler eine Liste von Abhängigkeiten, die ins Makefile übernommen werden können:

```
prompt% gcc -M love.c
love.o: love.c darling.h
```

Einfacher geht es mit dem Programm `makedepend`:

```
depend:
        makedepend -- $(CFLAGS) -- $(SRC_FILES)
```

Es fügt an das Ende des Makefiles die fehlenden Abhängigkeiten ein:

```
love.o : love.c
        $(CC) love.c

depend:
        makedepend -- $(CFLAGS) -- $(SRC_FILES)

# DO NOT DELETE

love.o: darling.h
```

Zusammen mit der ersten Abhängigkeit (love.o : love.c) wird jetzt love.c neu übersetzt, wenn sich love.c oder darling.h ändert.

6.6 Globale Definitionen (include-Anweisung)

Viele Makefiles innerhalb verschiedener Verzeichnisse eines Projekts sehen sich in großen Teilen ähnlich: es werden die gleichen **CFLAGS** definiert, der gleiche Compiler aufgerufen, die gleichen Suffix-Regeln verwendet, usw. Was liegt näher, als diese Gemeinsamkeiten in einer gemeinsamen Datei zu verwalten?

Glücklicherweise kennen GNU-`make` und viele andere `make`-Varianten eine **include**-Anweisung, mit der diese gemeinsame Datei eingebunden werden kann:

```
include common.mk
```

Hiermit wird die Datei common.mk eingebunden. Syntaktisch sieht das ganze dann so aus, dass diese Datei hier an diese Stelle hineinkopiert wird.

Bei der Verwendung der **include**-Anweisung ist darauf zu achten, das include am Zeilenanfang steht und dahinter mindestens ein Leerzeichen oder Tabulator-Zeichen folgt.

7 Richtlinien

Viele der Richtlinien in diesem Kapitel sind aus den "Makefile Conventions" für GNU-Programme entnommen (siehe http://www.gnu.org/prep/standards_50.html). Über das `automake`-Kommando könnnen Makefiles erzeugt werden, die diese Richtlinien unterstützen.

7.1 Allgemeine Konventionen

7.1.1 Standard-Shell

Normalerweise ist die Bourne-Shell (/bin/sh) als Standard-Shell vordefiniert. Es gibt aber auch `make`-Varianten, die die Standard-Shell über die Umgebungs-Variable **SHELL** vererbt bekommen. Um Probleme zu vermeiden, sollte daher die Bourne-Shell über folgendes Makro

```
SHELL = /bin/sh
```

als Standard-Shell definiert werden.

7.1.2 Suffix-Liste

Die Suffix-Liste sollte explizit gesetzt werden:

```
.SUFFIXES:
.SUFFIXES: .c .o
```

Die erste Zeile löscht die Suffix-Liste, die zweite Zeile setzt dann explizit die gewünschten Endungen.

Grund: Verschiedene `make`-Varianten haben unterschiedliche und inkompatible Suffix-Listen. Dies kann zu Verwirrungen auf unterschiedlichen Systemen führen.

7.1.3 Programmstart

Programme im Arbeitsverzeichnis müssen mit `./programm` gestartet werden.

Grund: Das aktuelle Verzeichnis ist nicht immer im Suchpfad enthalten.

7.2 Utilities

7.2.1 Shell-Kommandos

Auch wenn manche `make`-Varianten sowohl Korn-Shell- als auch Bash-Syntax verstehen, sollte man nur die Bourne-Shell-Syntax verwenden.

7.2.2 Standard-Kommandos

Folgende Unix-Kommandos können direkt verwendet werden:

cat cmp cp diff echo egrep expr false grep install-info ln ls mkdir mv pwd rm rmdir sed sleep sort tar test touch true

Bei den Optionen sollte man sich auf die gängigen Optionen beschränken, die auf allen Systemen vorhanden sind.

Grund: diese Kommandos sind auf allen Linux- und Unix-Varianten vorhanden

7.2.3 Kommandoaufrufe

Alle übrigen Kommandos wie Compiler-Aufruf und andere Programme sollten in Variablen abgespeichert werden.

Grund: Zum einen ist dies änderungsfreundlicher, zum anderen kann bei Bedarf das Kommando über die Kommandozeile mitgegeben werden.

7.3 Variablen

7.3.1 Kommando-Variablen

Nach Möglichkeit sollten Variablen genauso wie das Kommando heissen und mit dem Namen des Kommandos vorbelegt sein.

Beispiel:

```
YACC = yacc
```

7.3.2 Vordefinierte Variablen

Variable	Wert	Beschreibung
AR	ar	Archiver (zum Bibliotheken bauen)
CC	cc	C-Compiler
CXX	g++	C++-Compiler
CPP	$(CC) -E	C-Präprozessor
FC	f77	Fortran-Compiler
LD	ld	Linker, Loader
LEX	lex	lexikalische Analyse
MAKE	make	make-Kommando
PC	pc	Pascal-Compiler
YACC	yacc	Parser-Generator

Tipp: Bei der Verwendung von GNU-Make können die vordefinierten Variablen über

```
user@linux / $ make -p -f /dev/null | grep " = "
```

ausgegeben werden.

7.3.3 Flags

Flags zu Kommandos sollten in einer Variable mit dem Namen des Kommandos und der Endung **FLAGS** gekennzeichnet werden (s. Tabelle: **Flag-Variablen**)

Kommando	Flags	Beispiel	Bemerkung
AR	ARFLAGS	rv	
CC	CFLAGS	-g	Debug-Flag
CXX	CXXFLAGS		
CPP	CPPFLAGS	-DGERMAN	
FC	FFLAGAS		
LD	LDFLAGS	-lm	math. Bibliothek
LEX	LEXFLAGS		
MAKE	MAKEFLAGS	-k	
PC	PFLAGS		
YACC	YFLAGS	-v	Verbose-Flag

Abweichend von der obigen Namensgebung werden die Flags für den C-Compiler (**CC**) mit **CFLAGS**, für den Fortran-Compiler (**FC**) mit **FFLAGS**, für den Pascal-Compiler (**PC**) mit **PFLAGS** und für yacc (**YACC**) mit **YFLAGS** benannt. Dies hat historische Gründe.

7.3.4 Lebenswichtige Flags

Optionen, die für die Kompilation bzw. Erzeugung des Ziels unbedingt notwendig sind, werden nicht in diesen Variablen abgespeichert.

Grund: Es sollte weiterhin möglich sein, Variablen über den `make`-Aufruf oder über Umgebungs-Variablen zu setzen, ohne dass die Kompilation schief läuft.

Beispiel:

```
hugo.o : hugo.c
        $(CC) -c $(CPPFLAGS) $(CFLAGS) hugo.c
```

7.3.5 Install-Variablen

Die Variable **INSTALL** muss in jedem Makefile definiert sein und zum Installieren von Dateien dienen.

Daneben sollten die Variablen **INSTALL_PROGRAM** und **INSTALL_DATA** definiert werden, die zur Installation von Programmen und Daten dienen. Der Standard-Wert dafür sollte **$(INSTALL)** sein.

Name	Standard	Anmerkung
INSTALL	install	
INSTALL_PROGRAM	$(INSTALL)	Installation von Programmen
INSTALL_DATA	$(INSTALL) -m 644	Installation von Daten

Für die Installation von Programmen und Dateien sollte immer der komplette Dateiname und nicht nur der Verzeichnisname verwendet werden.

Beispiel:

```
install:
        $(INSTALL_PROGRAM) love $(bindir)/love
        $(INSTALL_DATA) loveletter $(datadir)/loveletter
```

7.4 Installations-Verzeichnis

Installations-Verzeichnisse sollten immer in Variablen abgelegt werden, so dass sich die Installation auch leicht an andere Zielverzeichnisse anpassen lässt.

Standardnamen für solche Variablen werden in diesem Abschnitt beschrieben. Sie basieren auf Standard-Dateisystemen von SVR4, 4.4BSD, Linux, Ultrix V4 und anderer moderner Betriebssysteme.

7.4.1 Root-Installations-Verzeichnis

Aus folgenden zwei Varablen sollten alle weitere Installations-Variablen abgeleitet werden:

prefix: Diese Variable enthält das Basis-Verzeichnis.
exec_prefix: Diese Variable enthält das Basis-Verzeichnis für einige ausführbare Programme. Als Standard-Wert enthält diese Variable den Wert von prefix.

Variable	Standard	Beschreibung
bindir	$(exec_prefix)/bin	hier werden die ausführbaren Programme für den Benutzer abgelegt
sbindir	$(exec_prefix)/sbin	hier werden die Programme für den System Administrator abgelegt
libexecdir	$(exec_prefix)/libexec	hier werden die Programme abgelegt, die von anderen Programmen benötigt werden
datadir	$(prefix)/share	für Architektur-unabhängige Daten-Dateien, die nicht verändert werden (read-only)
sysconfdir	$(prefix)/etc	Konfigurations-Dateien (read-only), die zu einer Single-Maschine gehören
sharedstatedir	$(prefix)/com	Architektur-unabhängige Daten-Dateien, die vom Programm verändert werden können
localstatedir	$(prefix)/var	lokale Architektur-unabhängige Daten-Dateien, die vom Programm verändert werden können
libdir	$(exec_prefix)/lib	Objekt- und Bibliotheks-Dateien (keine ausführbaren Programme)
infodir	$(prefix)/info	Info-Dateien
lispdir	$(datadir)/emacs/site-lisp	Emacs-Lisp-Dateien
includedir	$(prefix)/include	Header-Dateien
mandir	$(prefix)/man	Verzeichnis für die Manpages
man1dir	$(mandir)/man1	"1"er-Manpage
man2dir	$(mandir)/man2	"2"er-Manpage
...	...	...

manext	.1	Manpage Erweiterung
man1ext	.1	"1"er-Manpage Erweiterung
man2ext	.2	"2"er-Manpage Erweiterung
...	...	...
srcdir	-	Verzeichnis, in dem die Sourcen kompiliert werden

7.5 Standard Targets

7.5.1 Notwendige Targets

all	Kompilation des gesamten Programms (möglichst mit der Option "-g") sollte das Standard-Ziel sein
install	Kompilation und Installation des Programms, der Bibliotheken, usw...
uninstall	"install" wieder rückgängig machen
install-strip	Installation mit ge-"strip"-ten Programmen (strip entfernt die Symboltabelle aus einem Programm)
clean	Löschen aller Dateien, die beim Erstellen des Programms erzeugt werden
distclean	lösche alle Dateien, die nicht mehr benötigt werden
dist	Distributions-Tarfile erstellen; die Tar-Datei sollte ein Unterverzeichnis mit dem Werkzeugname und Versionsnummer enthalten, in dem sämtliche Programme und Dateien enthalten sind (zum Beispiel love-1.0.1)
check	Selbsttest (Überprüfung des Programms)

7.5.2 Nützliche Targets

mostlyclean	wie "clean", ausser Bibliotheken und andere Dateien, die zeitintensiv zu erstellen sind
TAGS	Erstellen/Update einer Tags-Tabelle
info	Info-Dateien erstellen
installcheck	Installation von Test-Dateien und -Verzeichnissen, die für "check" benötigt werden
installdirs	Verzeichnisse, die für "install" erzeugt werden müssen