Freie Universität Berlin

Makefiles

Automatisierung des Build-Prozesses

Freie Universität Berlin

Timeline

1. Einführung
2. Schreiben von Makefiles
3. Dependency-Generation etc.

Freie Universität Berlin

Der Build-Prozess

Aufbau:

1. Übersetzung der Quelldateien in Objektdateien
2. Linken der Objektdateien zu ausführbaren Dateien / Bibliotheken
3. Einbinden von externen Bibliotheken
4. (optional) automatisches Erkennen von geänderten Quelldateien bzw. nur partielles Rekompilieren

Freie Universität Berlin

make

- 1977 entwickelt, seit dem mehrere Implementationen
- „Kochrezept“
- In POSIX standardisiert, unter Linux setzen wir jedoch **GNU Make** ein, welches deutlich mehr Features hat

Freie Universität Berlin

make

- 1977 entwickelt, seit dem mehrere Implementationen
- „Kochrezept“
- In POSIX standardisiert, unter Linux setzen wir jedoch **GNU Make** ein, welches deutlich mehr Features hat
- Bei größeren Projekten lässt man die sog. **Makefiles** oft automatisch generieren.
- *Wir schreiben unsere Makefiles von Hand!*

Freie Universität Berlin

Regeln I: Grundlagen

Ein schlechtes Beispiel:

```
1 main: # Target
2     gcc -std=c99 -Wall -Wextra -pedantic main.c -o main # Command
```

Target:

- „Name“ der Regel
- i. d. R. die Datei die durch das Command erstellt wird
- Kann mit `$ make <Target>` ausgeführt werden

Freie Universität Berlin

Regeln II: Variablen

Ein (etwas) besseres, äquivalentes Beispiel:

```
1 CC = gcc
2 CFLAGS = -std=c99 -Wall -Wextra -pedantic
3 main:
4     $(CC) $(CFLAGS) main.c -o main
```

Standardvariablen:

CC	Der C-Compiler, i. d. R. cc, nur Überschreiben wenn nötig!
CFLAGS	Parameter für Aufruf eines C-Compilers
CPPFLAGS	Parameter für den C-Präprozessor
CXXFLAGS	Parameter für Aufruf eines C++-Compilers

Freie Universität Berlin

Regeln III: Voraussetzungen

Ein (wieder etwas) besseres Beispiel:

```
1 CFLAGS = -std=c99 -Wall -Wextra -pedantic
2 main: main.c # Prerequisite
3     $(CC) $(CFLAGS) main.c -o main
```

Voraussetzungen:

- „Zutaten“ für die Regel
- Falls diese sich ändern, gilt das Target als veraltet und wird neu kompiliert

Freie Universität Berlin

Regeln IV: Automatische Variablen

Ein (noch etwas) besseres Beispiel:

```
1 CFLAGS = -std=c99 -Wall -Wextra -pedantic
2 main: main.c
3     $(CC) $(CFLAGS) $^ -o $@ # Automatic Varibles
```

Automatische Variablen:

- $@ Name des Targets
- $^ Alle Prerequisites
- $+ Ähnlich, jedoch Mehrfachlistungen möglich und Reihenfolge wird eingehalten
- $< Erste Prerequisite

Freie Universität Berlin

Regeln V: Pattern-Regeln & Abhängigkeiten

Ein (noch) besseres Beispiel:

```
1 CFLAGS = -std=c99 -Wall -Wextra -pedantic
2 all: main # Default
3 %.o: %.c # Pattern for object files
4     $(CC) $(CFLAGS) $< -o $@
5 %: %.o # Pattern for linking executable
6     $(CC) $(LDFLAGS) $^ -o $@
7 .PHONY: all # all is a special rule (does not produce file "all")
```

Pattern & mehrere Regeln:

- Erste Regel (Standard) hat als Voraussetzung, dass `main` existiert
- Die Regel dafür setzt voraus, dass `main.o` existiert *und sich seit dem nicht geändert hat*
- Analog setzt `main.o` `main.c` voraus.

Freie Universität Berlin

Regeln VI: Implizite Regeln

Ein „perfektes“ (aber nicht komplettes) Beispiel:

```
1 CFLAGS = -std=c99 -Wall -Wextra -pedantic
2 all: main # Rule for main is implicit
3 .PHONY: all
```

Einige implizite Regeln:

- Objektdatei `%.o` aus `%.c`: `$(CC) $(CPPFLAGS) $(CFLAGS) -c`
- Ausführbare Datei `%` aus `%.o`: `$(CC) $(LDFLAGS) n.o $(LDLIBS)`

Freie Universität Berlin

Regeln VII: Kompletteres Beispiel

Ein kompletteres Beispiel:

```
1 CFLAGS = -std=c99 -Wall -Wextra -pedantic
2 LDFLAGS = -lm # Link with libm.a (maths)
3 all: main
4 main: test.o # additional object used for main, ie. main = main.o + test.o
5
6 clean: # Delete build files
7     @$(RM) -v *.o # @: Silence output, -v: list removed files
8 .PHONY: all clean
9 .PRECIOUS: *.o # Don't delete intermedita *.o files
```

Weiteres:

- Header?
- Unterverzeichnisse?
- Anderes Verzeichnis für Binaries?

Freie Universität Berlin

Header I

Problem:

- Header sind eigentlich auch „Zutaten“, deren Änderung die Kompilate veralten lassen können
- Wenn eine Quelldatei einen neuen Header inkludiert müssen wir das Makefile erneut anpassen

⇒ Präprozessor soll uns sagen, welche Header inkludiert wurden – `make` testet dann auf Änderungen

Header II

Präprozessoroptionen (GNU Toolchain):

-M	Gebe Make-Regeln auf der Konsole aus
-MM	Selbiges, ohne Systemheader
-MD	Kompiliere ganz normal, schreibe die Informationen in Datei
-MMD	Selbiges, ohne Systemheader

-MF <Datei> In Kombination mit Obigem: Schreibe in diese Datei

Beispiel:

```
1 $ gcc -MM main.c
2 main.o: main.c test.h
```

Anmerkung: Ab hier könnte es ggf. sinnvoll sein, gcc oder clang festzuschreiben

Freie Universität Berlin

Header III

„Vollständiges“ Beispiel:

```
1  CFLAGS = -std=c99 -Wall -Wextra -pedantic
2  CPPFLAGS = -MMD -MF $*.d # Generate dependency files
3  LDLIBS = -lm
4  all: main
5
6  -include *.d # Include dependency files, ignore if non-existant
7
8  main: test.o
9  %.o: %.d # If dependencies have changed, recompile too
10 clean:
11     @$(RM) -v main *.o *.d
12 .PHONY: all clean
13 .PRECIOUS: *.o *.d
```

Unterverzeichnisse I

Modularisierung:

- Bei großen Projekten i. d. R. viele Unterverzeichnisse
- Idee: Pro Verzeichnis ein Makefile; jedes Unterverzeichnis wird von dem darüberliegenden eingebunden
- Legen eine (oder mehrere) Liste(n) aller zu kompilierenden Objekte an
- Diese werden dann als Abhängigkeit zu der Applikation eingetragen.

Unterverzeichnisse II

Haupt-Makefile:

```
1  CFLAGS = -std=c99 -Wall -Wextra -pedantic
2  CPPFLAGS = -MMD -MF $*.d
3
4  all: main
5
6  include mod1/Makefile
7
8  main: $(OBJS) # build main from all OBJS (added in subdirs)
9  %.d: ; # if a dependency file is missing (eg. on first run), don't panic
10 clean:
11     # delete object files and their dependencies as well as the executable
12     @$(RM) -v $(OBJS) $(OBJS:.o=.d) main
13 .PHONY: all clean
14 .PRECIOUS: *.d *.o
```

Freie Universität Berlin

Unterverzeichnisse II.b

Unter-Makefile:

```
1 OBJS += mod1/mod1.o
2 include mod1/*.d
```

- Variable `OBJS` enthält alle Objekte für das Programm
- Sie wird in den Unterverzeichnissen gesetzt, bzw. erhält dort jeweils neue Elemente
- Ein Unterverzeichnis kann wiederum ein weiteres einbinden

Freie Universität Berlin

Referenzen I

- Free Software Foundation.
  GNU make Manual

  https://www.gnu.org/software/make/manual/make.html#Implicit-Variables

  https://www.gnu.org/software/make/manual/make.html#Automatic-Variables

  https://www.gnu.org/software/make/manual/make.html#Pattern-Match

  https://www.gnu.org/software/make/manual/make.html#Catalogue-of-Rules

  https://www.gnu.org/software/make/manual/make.html#Automatic-Prerequisites

- Wikipedia Autoren.
  Artikel der englischen Wikipedia

  https://en.wikipedia.org/wiki/Make_(software)