Zur Nutzung von makefile-Dateien

Lukas C. Bossert

circleCI-Test

Größere LaTeX-Projekte mit vielen Dateien zu managen ist nicht immer einfach und man muss nach dem Übersetzen verschiedene Schritte ggf. manuell ausführen und das PDF weiter verarbeiten. Beispielsweise wenn man das PDF zusätzlich noch in einer komprimierten Fassung haben möchte oder wissen muss, auf welchen Seiten Farbinformationen im PDF hinterlegt sind.

Die folgenden Ausführungen beziehen sich auf praxisnahe Funktionsweisen des Programms GNUmake (Unix/macOS). Für andere Betriebssysteme gibt es entsprechende Varianten (bspw. für Windows nmake).

Mittels einer makefile-Datei und dem Programm make können mehrere Befehle gleichzeitig und/oder hintereinander ausgeführt werden, sodass verschiedene händische Arbeitsschritte abgenommen werden können.[1]

Zunächst erläutere ich die Eigenschaften und den Aufbau einer makefile-Datei. Anschließend zeige ich an kleinen Beispielen, worin das Potenzial dieser unscheinbaren Datei liegt. Mir ist weniger daran gelegen, die (durchaus komplexe) Logik bei den Abhängigkeiten von »Ziel« und »Quelle« (s. u.) zu durchdringen [2] als vielmehr einen praktisch orientierten Einblick zu geben.

Der Aufbau einer minimalen makefile-Datei

Eine makefile-Datei ist eine schlichte Textdatei *ohne* Endung. Sie liegt idealerweise im gleichen Ordner wie die Hauptdatei des LaTeX-Projekts. Sie kann mit Variablen arbeiten und man kann alle Befehle ausführen lassen, die man auch im Terminal eingeben kann. Dies sind die zwei wichtigsten Merkmale, die wir gleich nutzen werden.

Zunächst definieren wir die Variable PROJECT, die den Dateinamen der Hauptdatei unseres LaTeX-Projekts beinhaltet.

```
1 PROJECT  = dtk-make-bossert
```

Nun wollen wir den Arbeitsschritt zum Erstellen der PDF einbauen.

```
1 all:
2     lualatex $(PROJECT)
```

[1] Dieser Beitrag stellt eine Ergänzung zu »make – nur etwas für Profis?« [1] dar, da es hierbei um konkrete Beispiele (m)eines LaTeX-Alltags geht.

Mit `all` wird ein »Ziel« angegeben. In diesem Fall ist es die Standardausführung, wenn keine weiteren Angaben beim Ausführen der `makefile`-Datei gemacht werden. Alle folgenden Zeilen, die zu diesem »Ziel« gehören, werden mit einem Tab eingerückt. Mit `lualatex $(PROJECT)` wird die oben definierte Variable aufgerufen, sodass `lualatex dtk-make-bossert` ausgeführt wird.

In der Reinform sieht ein Befehl also in etwa so aus.

```
Ziel: Quelle(, ..., Quelle)
    Befehl1
    Befehl2
    .
```

Um die `makefile`-Datei auszuführen, navigiert man im Terminal zum Hauptordner des LaTeX-Projects und führt lediglich den Befehl `make` aus.

Weitere Variablen und Arbeitsanweisen in der `makefile`-Datei

Nach dieser kurzen Einführung können wir verschiedene »Ziele« basteln, um sie bei Bedarf oder immer ausführen zu lassen.

Es empfielt sich anzugeben, wo `make` die Shell findet. Dies erfolgt mit einer Variable.

```
SHELL = bash
```

Anschließend führen wir noch ein paar Farben ein, um die Lesbarkeit der Informationsdarstellung zu erhöhen.

```
# Colors
RED   = \033[0;31m
CYAN  = \033[0;36m
NC    = \033[0m # No color
echoPROJECT = @echo -e "$(CYAN) ‹$(PROJECT)›"
```

Die letzte Variable gibt im Terminal den Projektnamen farblich aus.

Jetzt kommt noch die Definition von `PHONY`-Zielen [3, S. 13–15]. Anhand dieser Wortliste weiß `make`, dass es sich hierbei nicht um Dateinamen handelt, sondern um auszuführende »Ziele«.

```
.PHONY: all article zip
```

Als erstes »Ziel« definieren wir die Erstellung des Artikels, wofür wir eine weitere Variable nutzen, die das aktuelle Datum abruft.

```
DATE  = $(shell /bin/date "+%Y-%m-%d")
```

Jetzt das »Ziel« selbst.

```
1  article:
2    $(echoPROJECT) "* compiling article * $(NC)"
3    latexmk -lualatex -quiet -f -cd -view=pdf -output-directory=tmp $(PROJECT).
   ↪tex
4    @cp tmp/$(DATE)/$(PROJECT).pdf .
5    $(echoPROJECT) "* article compiled * $(NC)"
```

Als erstes soll im Terminal angezeigt werden, welches »Ziel« von `make` gerade ausgeführt wird (Z.2) bzw. abgeschlossen wurde (Z.6). Anschließend wird das PDF mittels `latexmk` erstellt, wozu weitere Optionen angegeben sind: Um den Hauptordner von allen temporären Dateien frei zu halten, werden diese in ein separates Verzeichnis erstellt.

Das PDF wird schließlich in den Hauptordner kopiert (Z. 5). Mit dem Präfix `@` wird die auszuführende Befehlszeile nicht im Terminal angezeigt, lediglich deren Result. Mit `make article` lässt sich diese Passage direkt ansteuern und ausführen.

Besonders bei bildlastigen PDFs ist deren Dateigröße manchmal auch zu groß, um sie für Korrekturen etc. zu verschicken. Das PDF muss dann in einem weiteren Schritt komprimiert werden. Dieser Vorgang lässt sich ebenfalls von `make` mittels Ghostscript ausführen.[2]

Das »Ziel« ist `minimize` und als »Quelle« geben wir das oben formulierte »Ziel« `article` an. Das heißt, dass beim Aufruf von `minimize` zuerst das »Ziel« `article` ausgeführt wird – Dank `latexmk` wird nur bei veränderter `tex`-Datei neu übersetzt. Somit wird gewährleistet, dass immer die neuste PDF-Version minimiert wird.

```
1  minimize: article
2    $(echoPROJECT) "* minimizing article * $(NC)"
3    @-mkdir archive
4    @rm -f archive/$(PROJECT)-$(DATE)*.pdf
5    gs \
6    -sDEVICE=pdfwrite \
7    -dCompatibilityLevel=1.4 \
8    -dPDFSETTINGS=/printer \
9    -dNOPAUSE \
10   -dQUIET \
11   -dBATCH \
12   -sOutputFile=archive/$(PROJECT)-$(VERS).pdf \
13   $(PROJECT).pdf
14   $(echoPROJECT) "* article minimized * $(NC)"
```

[2] Zu den einzelnen Optionen des Ghostscriptbefehls s. XXX

Zunächst wird ein Ordner `archive` erstellt (Z. 3). Sollte dieser Ordner bereits existieren, wirft `make` zwar einen Fehler, dieser wird jedoch dank des vorangestellten – bei `mkdir` nicht zum Abbruch des Skripts führen. In Zeile 4 wird ggf. eine ältere PDF gelöscht. Das PDF wird nun mit Ghostscript komprimiert (Z. 5 ff.) und mit Datumsangabe im Dateinamen im Ordner `archive` abgelegt.

Um auch zugleich den Status quo des LaTeX-Projects festzuhalten, kann man alle notwendige Dateien tagesaktuell zippen. Somit hat man immer den letzten Tagesstand im Ordner `archive` gesichert. Dafür bedarf es noch ein paar Variablen, die wir vorweg definieren.

```
1 # zip
2 PWD   = $(shell pwd)
3 TEMP := $(shell mktemp -d -t tmp.XXXXXXXXXX)
4 TDIR  = $(TEMP)/$(PROJECT)
5 VERS  = $(shell /bin/date "+%Y-%m-%d---%H-%M-%S")
6 DATE  = $(shell /bin/date "+%Y-%m-%d")
```

Das »Ziel« heißt `zip` und es wird wiederum zuerst `article` ausgeführt, um die aktuelle Projektversion zu zippen.

```
1 zip: article
2   $(echoPROJECT) "* zipping files * $(NC)"
3   @-mkdir archive
4   @rm -f archive/$(PROJECT)-$(DATE)*.zip
5   @mkdir $(TDIR)
6     @cp $(PROJECT).{bib,tex,pdf,csv} README.md makefile $(TDIR)
7   @cd $(TEMP); \
8    zip -Drq $(PWD)/archive/$(PROJECT)-$(VERS).zip $(PROJECT)
9   $(echoPROJECT) "* files zipped * $(NC)"
```

In Zeile 3 wird wiederum zuerst ein Ordner `archive` erstellt, in den später die gezippte Datei abgelegt wird. Mit Zeile 4 wird die tagesaktuelle Datei gelöscht, sodass für jeden Tag immer nur eine und die letzte Version in `archive` abgelegt wird.

In den folgenden Zeilen wird der Packvorgang ausgeführt, zunächst erfolgt die Erstellung eines temporären Ordners, anschließend werden die zu zippenden Dateien ausgewählt und schließlich die `zip`-Datei im Ordner `archive` abgelegt.

In Zeile 6 ist eine sehr effiziente Syntax von `make` eingebaut:

```
$(PROJECT).{bib,tex,pdf,csv}
```

Dies ist gleichbedeutend mit

```
$(PROJECT).bib $(PROJECT).tex $(PROJECT).pdf $(PROJECT).csv
```

Die kommaseparierten Werte in den geschweiften Klammern werden expandiert und mit (in diesem Fall) dem Präfix gekoppelt. Damit erspart man sich manche Tipparbeit.[3]

Möchte man sein PDF an eine Druckerei geben, braucht man die genaue Anzahl der Farbseiten plus der Auflistung der Farben. Es wäre fatal (und unnötig), dies bei größeren PDFs von Hand zu tun. Folgende Code gibt eine Tabulator getrennte csv-Datei mit der prozentualen Farbabdeckung von jeder Seite.[4] Damit kann man sehr leicht erkennen, ob das CMYK-Farbmodell korrekt ist und auf welchen Seiten Cyan, Magenta oder Gelb (CMY) verwendet wird.

```
1  count.colorpages:
2    $(echoPROJECT) "* listing and counting colored pages * $(NC)"
3    @echo "Meta information about colors in $(PROJECT)"
4    @gs -o - -sDEVICE=inkcov $(PROJECT).pdf \
5     | tail -n +5 \
6     | sed '/^Page*/N;s/\n//' \
7     | tee $(PROJECT).csv
8    @echo -n "List of pages with colors: "
9    @cat $(PROJECT).csv \
10    | awk '$$3!="0.00000" ||  $$4!="0.00000" || $$5!="0.00000"{if(length(
   ↪colored))colored=colored","$$2;else colored=$$2} END{print colored}' \
11    | tee  -a $(PROJECT).csv
12   @echo -n "Total amount of pages with color: "
13   @gs -o - -sDEVICE=inkcov $(PROJECT).pdf \
14    | grep -v "^ 0.00000  0.00000  0.00000" \
15    | grep "^ " \
16    | wc -l \
17    | sed 's/[[:space:]]//g' \
18    | tee  -a $(PROJECT).csv
19   $(echoPROJECT) "* colored pages listed and counted * $(NC)"
```

In Zeile 3 rufen wir Ghostscript auf lassen die Farbabdeckung jeder Seite ausgeben. Anschließend (Z. 4) werden die ersten fünf Zeilen dieser Liste gelöscht (es sind für unser Vorhaben nicht notwendige Metadaten), und schließlich (Z. 5) ein unschöner Absatz entfernt, sodass in Zeile 6 das Speichern einer csv-Datei ausgeführt wird.

Schließlich werden alle Farbseiten kommaseparierte aufgelistet und ebenfalls in der csv-Datei ergänzt (Z.8–11).

[3] In diesem konkreten Fall wäre `$(PROJECT).*` einfacher gewesen, würde aber die spezielle Syntax nicht zeigen.

[4] `https://stackoverflow.com/a/28369599`

Mit dem nochmaligen Aufruf von Ghostscript in Zeile 12 und der direkten Weiterverarbeitung im Suchen/Ersetzen-Prinzip (`grep`), wird die Gesamtzahl der Farbseiten ermittelt. Diese Zahl wird in die letzte Zeile der `csv`-Datei geschrieben.

Die `csv`-Datei mit der Liste der Farbseiten für diesen Artikel sieht dann so aus.

```
Page 1 0.00000  0.00000  0.00000  0.12394 CMYK OK
Page 2 0.00000  0.00000  0.00000  0.26883 CMYK OK
Page 3 0.00000  0.00000  0.00000  0.34570 CMYK OK
Page 4 0.00000  0.00000  0.00000  0.32757 CMYK OK
Page 5 0.00016  0.00035  0.00017  0.33428 CMYK OK
Page 6 0.00000  0.00000  0.00000  0.46510 CMYK OK
Page 7 0.00000  0.00000  0.00000  0.63213 CMYK OK
Page 8 0.00000  0.00000  0.00000  0.36116 CMYK OK
5
1
```

Der Einsatz von `make`

Damit haben wir nun ein paar hilfreiche »Ziele« und Vorgehensweisen kennengelernt, die wir nun in eine `makefile`-Datei schreiben.[5]

```
PROJECT  = dtk-make-bossert
SHELL = bash
MAKE  = make
# zip
PWD   = $(shell pwd)
TEMP := $(shell mktemp -d -t tmp.XXXXXXXXXX)
TDIR  = $(TEMP)/$(PROJECT)
VERS  = $(shell /bin/date "+%Y-%m-%d---%H-%M-%S")
DATE  = $(shell /bin/date "+%Y-%m-%d")
# Colors
RED   = \033[0;31m
CYAN  = \033[0;36m
NC    = \033[0m
echoPROJECT = @echo -e "$(CYAN) <$(PROJECT)>$(RED)"

.PHONY: all article zip

# default
all:
  $(MAKE) article
```

[5] Diese `makefile`-Datei ist auch online verfügbar: `https://github.com/LukasCBossert/dtk-make/blob/master/makefile`.

```
21  $(MAKE) minimize
22  $(MAKE) zip
23  $(MAKE) count.colorpages
24  $(echoPROJECT) "* all files processed * $(NC)"
25
26 # compile article
27 article:
28  $(echoPROJECT) "* compiling article * $(NC)"
29  latexmk \
30   -lualatex \
31   -quiet \
32   -view=pdf \
33   -output-directory=tmp \
34   $(PROJECT).tex
35  @cp tmp/$(PROJECT).pdf .
36  $(echoPROJECT) "* article compiled * $(NC)"
37
38 # zip files for sending etc.
39 zip: article
40  $(echoPROJECT) "* start zipping files * $(NC)"
41  @-mkdir archive
42  @rm -f archive/$(PROJECT)-$(DATE)*.zip
43  @mkdir $(TDIR)
44  @cp $(PROJECT).{bib,tex,pdf,csv} README.md makefile $(TDIR)
45  @cd $(TEMP); \
46   zip -Drq $(PWD)/archive/$(PROJECT)-$(VERS).zip $(PROJECT)
47  $(echoPROJECT) "* files zipped * $(NC)"
48
49 # minimize PDF
50 minimize: article
51  $(echoPROJECT) "* minimizing article * $(NC)"
52  @-mkdir archive
53  @rm -f archive/$(PROJECT)-$(DATE)*.pdf
54  gs \
55   -sDEVICE=pdfwrite \
56   -dCompatibilityLevel=1.4 \
57   -dPDFSETTINGS=/printer \
58   -dNOPAUSE \
59   -dQUIET \
60   -dBATCH \
61   -sOutputFile=archive/$(PROJECT)-$(VERS).pdf \
62  $(PROJECT).pdf
63  $(echoPROJECT) "* article minimized * $(NC)"
64
```

```
# count pages with colors
# › https://stackoverflow.com/a/28369599
# › https://stackoverflow.com/q/54991314
count.colorpages:
  $(echoPROJECT) "* listing and counting colored pages * $(NC)"
  @echo "Meta information about colors in $(PROJECT)"
  @gs -o - -sDEVICE=inkcov $(PROJECT).pdf \
    | tail -n +5 \
    | sed '/^Page*/N;s/\n//' \
    | tee $(PROJECT).csv
  @echo -n "List of pages with colors: "
  @cat $(PROJECT).csv \
    | awk '$$3!="0.00000" ||  $$4!="0.00000" || $$5!="0.00000"{if(length(
 ↪colored))colored=colored","$$2;else colored=$$2} END{print colored}' \
    | tee  -a $(PROJECT).csv
  @echo -n "Total amount of pages with color: "
  @gs -o - -sDEVICE=inkcov $(PROJECT).pdf \
    | grep -v "^ 0.00000  0.00000  0.00000" \
    | grep "^ " \
    | wc -l \
    | sed 's/[[:space:]]//g' \
    | tee  -a $(PROJECT).csv
  $(echoPROJECT) "* colored pages listed and counted * $(NC)"
```

Wie bereits erwähnt, wird das »Ziel« `all` ausgeführt (weil es an erste Stelle steht), wenn man im Terminal lediglich `make` eingibt. In unserer Datei werden alle »Ziele« nun standardmäßig ausgeführt.

Möchte man hingegen nur ein bestimmtes »Ziel« ausführen, kann man dieses mit `make ‹ZIEL›` direkt ansteuern, beispielsweise `make zip`.

Diese `makefile`-Datei lässt sich nun nach Belieben ergänzen und verändern, um auch Projekt spezifische Anforderungen in der Nachbearbeitung effizient zu bearbeiten.

Literatur und Software

[1] Rolf Niepraschk: »`make` – nur etwas für Profis?«, *DTK*, *13*.1 (2001), 54–58.

[2] Thomas Peschel-Findelsen: make. GE-PACKT, mit-p Verlag, Bonn, 2004.

[3] Mecklenburg Robert: Managing Projects with GNU Make, 3. Aufl., O'Reilly, Köln, 2005.