Zur Nutzung von `makefile`-Dateien

Lukas C. Bossert

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) "* making 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 (falls er schon existiert wird mit dem Präfix – zwar eine Fehlermeldung ausgegeben, diese führt jedoch nicht zum Abbruch des Befehls). Das PDF wird komprimiert und mit Datumsangabe 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} README.md makefile $(TDIR)
7   cd $(TEMP); \
8   zip -Drq $(PWD)/archive/$(PROJECT)-$(VERS).zip $(PROJECT)
9   $(echoPROJECT) "* files zipped * $(NC)"
```

In Zeile 6 ist eine sehr effiziente Syntax eingebaut:

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

Dies ist gleichbedeutend mit

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

Die kommaseparierten Werte in den geschweiften Klammern werden mit (in diesem Fall) dem Präfix gekoppelt. Somit spart man sich manche Tipparbeit.

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.[3] Damit kann man

[3] https://stackoverflow.com/a/28369599

sehr leicht erkennen, ob das CMYK-Farbmodell korrekt ist und auf welchen Seiten Cyan, Magenta oder Gelb (CMY) verwendet wird.[4]

```
count.colorpages: article
  $(echoPROJECT) "* counting colored pages * $(NC)"
  @ gs \
  -o - \
  -sDEVICE=inkcov \
  $(PROJECT).pdf \
  |tail -n +5 \
  |sed '/^Page*/N;s/\n//'\
  |sed -E '/Page [0-9]+ 0.00000 0.00000 0.00000 / d' \
  | tee  $(PROJECT).csv
  @echo -e "Total amount of pages with color: "
  @ gs -o - -sDEVICE=inkcov $(PROJECT).pdf | \
   grep -v "^ 0.00000  0.00000  0.00000" | grep "^ " | wc -l
  $(echoPROJECT) "* colored pages counted * $(NC)"
```

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

```
Page 1 0.00000  0.00000  0.00000  0.12851 CMYK OK
Page 2 0.00000  0.00000  0.00000  0.32921 CMYK OK
Page 3 0.00000  0.00000  0.00000  0.29300 CMYK OK
Page 4 0.00016  0.00035  0.00017  0.44022 CMYK OK
Page 5 0.00000  0.00000  0.00000  0.56766 CMYK OK
Page 6 0.00000  0.00000  0.00000  0.60892 CMYK OK
Page 7 0.00000  0.00000  0.00000  0.02244 CMYK OK
```

Der Einsatz von `make`

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

```
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")
```

[4] Man könnte diesen Code noch verbessern, indem man direkt eine kommaseparierte Liste ausgegeben bekommt, die die Seitenzahlen der Farbseiten aufzählt.

```
# 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
  $(MAKE) minimize
  $(MAKE) zip
  $(MAKE) count.colorpages
  $(echoPROJECT) "* all files processed * $(NC)"

# compile article
article:
  $(echoPROJECT) "* making article * $(NC)"
  latexmk -lualatex -quiet -f -cd -view=pdf -output-directory=tmp $(PROJECT).
 ↪tex
  @cp tmp/$(PROJECT).pdf .
  $(echoPROJECT) "* article compiled * $(NC)"

# zip files for sending etc.
zip: article
  $(echoPROJECT) "* start zipping files * $(NC)"
  @-mkdir archive
  @rm -f archive/$(PROJECT)-$(DATE)*.zip
  @mkdir $(TDIR)
  @cp $(PROJECT).{bib,tex,pdf} README.md makefile $(TDIR)
  cd $(TEMP); \
  zip -Drq $(PWD)/archive/$(PROJECT)-$(VERS).zip $(PROJECT)
  $(echoPROJECT) "* files zipped * $(NC)"

# count pages with colors › https://stackoverflow.com/a/28369599
count.colorpages: article
  $(echoPROJECT) "* counting colored pages * $(NC)"
  @ gs \
  -o - \
  -sDEVICE=inkcov \
  $(PROJECT).pdf \
  |tail -n +5 \
  |sed '/^Page*/N;s/\n//'\
```

```
53    |sed -E '/Page [0-9]+ 0.00000 0.00000 0.00000 / d' \
54    | tee  $(PROJECT).csv
55    @echo -e "Total amount of pages with color: "
56    @ gs -o - -sDEVICE=inkcov $(PROJECT).pdf | \
57     grep -v "^ 0.00000  0.00000  0.00000" | grep "^ " | wc -l
58    $(echoPROJECT) "* colored pages counted * $(NC)"
59
60  # minimize PDF
61  minimize: article
62    $(echoPROJECT) "* minimizing article * $(NC)"
63    @-mkdir archive
64    @rm -f archive/$(PROJECT)-$(DATE)*.pdf
65    gs \
66    -sDEVICE=pdfwrite \
67    -dCompatibilityLevel=1.4 \
68    -dPDFSETTINGS=/printer \
69    -dNOPAUSE \
70    -dQUIET \
71    -dBATCH \
72    -sOutputFile=archive/$(PROJECT)-$(VERS).pdf \
73    $(PROJECT).pdf
74    $(echoPROJECT) "* article minimized * $(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 nach Belieben ergänzen und verändern, um auch auf Projekt spezifische Anforderungen zu reagieren.

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.