Hardwarenahe Programmierung I

U. Kaiser, R. Kaiser, M. Stöttinger, S. Reith

(HTTP: http://www.cs.hs-rm.de/~kaiser
EMail: robert.kaiser@hs-rm.de)

Wintersemester 2021/2022

6. Weitere Werkzeuge

https://memegenerator.net/Swiss-Army-Knife-Of-Doom

Inhalt

Übersicht

- Bisher besprochen:
  - Werkzeuge zum Generieren / Analysieren / Konvertieren von Code
    - Compiler, Linker
    - `objdump`, `objcopy`, `nm`, ...
  - Werkeuge zum Erstellen von Quellcode
    - Editor
- In diesem Abschnitt:
  - Werkzeuge zur Versionskontrolle
  - Build-Werkzeuge (i.W. `make`)

Werkzeuge zur Versionskontrolle

- Tools zum Erfassen von Änderungen an Dateien
  - Alle Zwischenstände werden archiviert, können bei Bedarf wiederhergestellt werden
  - Meist (aber nicht ausschließlich) auf Programm-Quelltexte angewendet
- Typische Funktionen
  - Protokollieren von Änderungen („*wer hat wann was geändert*")
  - Wiederherstellen früherer Stände ($\rightarrow$ unerwünschte Änderungen können zurückgenommen werden)
  - Koordinieren von Zugriffen mehrerer Entwickler an einem Projekt
  - Parallele Entwicklung mehrerer Entwicklungszweige (*branches*), zusammenfügen (*merge*) von Zweigen
  - Versionsstände durch *Tags* markieren und wiederherstellen

Funktion

- Verschiedene Versionen von Dateien (i.d.R Quellcode → ASCII-Textdateien) werden einem *Repository* abgelegt
- Basisoperationen:
  - *Checkin*: Neue Version einer Datei zum Repository hinzufügen
  - *Checkout*: Lokale Arbeitskopie aus dem Repository erstellen
- Weitere Operationen:
  - Anlegen eines neuen Repository
  - Vergleich zwischen Arbeitskopie und Repository ggf. mit patch-Erzeugung
  - Zusammenführen von Zweigen

Ort des Repository

1. **Lokale Versionsverwaltung**: Jede Datei wird einzeln versioniert (z.B. SCCS, RCS, MS Word, ..)
2. **Zentrale Versionsverwaltung**: Repository liegt auf einem Server, Clients können über Netzwerk ein- und auschecken (z.B. CVS, SVN)
3. **Verteilte Versionsverwaltung**: Kein zentrales Repository mehr: Alle arbeiten auf eigenen lokalen Kopien, die mit jeder anderen Kopie abgeglichen können. Dazu exisiteren automatisierte „merge“-Mechanismen (z.B. Git, Mercurial)

Konzepte

Hochschule RheinMain

1. **Lock, Modify, Write**: Datei wird beim Checkout für andere Benutzer gesperrt, bei Checkin wieder freigegeben
   - \+ Kein nachträgliches Zusammenführen erforderlich
   - – Gleichzeitige Bearbeitung einer Datei nicht möglich
   - ▸ auch: „pessimistische Versionsverwaltung"
   - ▸ Bei Binärdateien zwingend erforderlich
2. **Copy, Modify, Merge**: Gleichzeitiges Bearbeiten durch mehrere Personen möglich: Änderungen werden automatisch oder manuell zusammengeführt
   - \+ Ermöglicht unabhängiges, verteiltes Entwickeln
   - – Problematisch bei nicht-Textdateien
     → ggf. Lock, Modify, Write für bestimmte Datei-Arten
   - ▸ auch: „optimistische Versionsverwaltung"

Beispiele (1)

- 1. Dinosaurier ...
  - SCCS (*Source Code Control System*)
    - ★ Erste Software dieser Art (Rochkind 1972)
    - ★ Betrachtet nur Einzeldateien
    - ★ Heute weitgehend bedeutungslos
  - RCS (*Revision Control System*)
    - ★ Ähnlich SCCS (Tichy 1985)
    - ★ „Delta-Kodierung": Nur Änderungen werden aufgezeichnet
    - ★ Betrachtet ebenfalls nur Einzeldateien
    - ★ Hat an Bedeutung verloren, wird aber immer noch weiterentwickelt
    - ★ Realisiert durch Einzelkommandos: `rcsintro`, `rcs`, `rcsfile`, `ci`, `co`, `rcsdiff`, `rlog`, `rcsmerge`, `rcsclean`, `rcsfreeze`

Beispiele (2)

Hochschule RheinMain

- 2. Oldies ...
  - CVS (*Concurrent Version System*)
    - ★ Nachfolger von RCS (gleiches Dateiformat)
    - ★ Ursprünglich: Netzwerk-Aufsatz für RCS
    - ★ Betrachtet Mengen von Dateien ($\rightarrow$ Projekte)
    - ★ Nach wie vor eigene Historie je Datei, Projektversionen als Sammlung von Dateien mit individuellen Versionen
    - ★ Zentrales Server-Repository
    - ★ Probleme mit Binärdateien, Verzeichnissen und Umbenennungen
    - ★ Auch heute noch vielfach verwendet
    - ★ Weiterentwicklung eingestellt
- 3. Aktuelle Tools ...
  - Subversion
    - ★ Nachfolger von CVS (aber völlig neue Implementierung)
    - ★ Ziel: Probleme von CVS vermeiden
    - ★ Projektbezogene Historie
    - ★ Zentrales Repository
    - ★ Weite Verbreitung

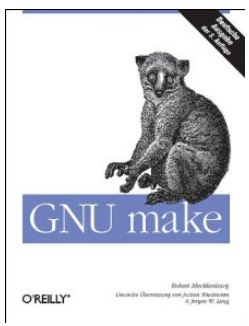
Beispiele (3)

- 3. Aktuelle Tools (Forts.) ...
  - Git (Linus Torvalds)
  - Mercurial (Matt Mackall)
  - Bazaar (Canonical Ltd.)
- Verteilte Versionsverwaltung
- Copy, Modify, Merge Konzept

Make

- Universelles Werkzeug zur Automatisierung von Abläufen
- Haupt- (aber nicht einzige) Anwendung: Übersetzen und Binden von C/C++-Programmen
- Dabei: nur die notwendigen Teile neu übersetzen/binden:
  - Quellcode oder include-Datei jünger als Objektdatei
  - Objektdatei oder Bibliothek jünger als ausführbare Datei
- Über Make kann man ganze Bücher schreiben ...

  R. Mecklenburg
  GNU make
  O'Reilly
  ISBN 3897214083
  328 Seiten



- ... hier nur das Wichtigste ...

Make-Kommando

- Kommando `make` sucht im aktuellen Verzeichnis nach einer Datei `Makefile`, falls die nicht gefunden wird, `makefile`
- Wichtige Optionen:
  - `-f <filename>`: Liest `<filename>` statt `Makefile`
  - `-C <verzeichnis>`: Wechselt zuvor in `<verzeichnis>`
- Weitere Möglichkeiten auf der Kommandozeile:
  - `make <target>`: Gezielter Aufruf eines „Targets" (s.u.) im Makefile (Per Default wird das erste Target aufgerufen)
  - `make <Makro>=<Wert>` Dem Makro wird ein Wert zugewiesen. Überschreibt ggf. Festlegungen im Makefile (Beispiel: `make CFLAGS=-g`)

Makefile

- Allgemeiner Aufbau: Einträge der Form

```
<target>: <depend>
    <Kommando>
    ...
```

- Darin sind:
  - `<target>`: Zu erzeugendes Objekt
  - `<depend>`: Liste der Objekte, von denen `<target>` abhängt
  - `<Kommando>`: Kommando(s) zum Erzeugen des `<target>`
    Achtung: müssen mit Tabulator eingerückt werden!
- Beispiel:

```
hello: hello.o
    gcc -o hello hello.o
```

Targets und Sub-Targets

- Abhängigkeitsliste kann weitere Targets aufrufen:

```
hello: hello.o
    gcc -o hello hello.o

hello.o: hello.c inc/hellodefs.h
    gcc -C -o hello.o hello.c
```

(Annahme: `hello.c` enthält `#include "inc/hellodefs.h"`)

- Tipp: Autogenerieren von Abhängigkeiten:

```
$ gcc -MM hello.c
hello.o: hello.c inc/hellodefs.h
```

(N.B.: Funktioniert auch bei verschachtelten `#includes`)

Automatische Abhängigkeiten

- Anwendung im Makefile:

```
hello.d: hello.c
	gcc -MM hello.c >hello.d

-include hello.d
```

- Abhängigkeiten automatisch aus Quellcode generieren und ins Makefile einfügen
  (N.B.: Das „-" vor `include` bedeutet: Datei einfügen, falls vorhanden. Falls nicht vorhanden: stillschweigend weiterarbeiten)

Spezielle Targets

- Viele Makefiles definieren üblicherweise Targets „all", „clean", etc.
  - `make all`: „Alles" bauen (oft synonym mit `make`)
  - `make clean`: Zwischendateien (z. B. `*.o`) löschen
  - `make clobber`, `make distclean`: Zwischen- und Ergebnisdateien [1] löschen (z.B. vor Einchecken oder Auslieferung)
  - `make install`: Erzeugte Dateien ins System installieren (erfordert i.d.R. root-Rechte)
- I.d.R „künstliche" (*phony*) Targets: Es wird keine Datei dieses Namens erzeugt → Problem, falls doch mal eine existiert:

```
$ touch all
$ make all
$ make: Fuer das Ziel »all« ist nichts zu tun.
```

- Abhilfe: Im Makefile *phony* Targets deklarieren:

```
.PHONY all clean clobber ...
```

[1] anders ausgedrückt: alles außer dem Quellcode

„Eingebaute" Makros

Hochschule RheinMain

Makro	Bedeutung
`$@`	**Voller Name des Target**
`$*`	**Name des Target ohne Dateiendung**
`$<`	Erste Abhängigkeit aus der Abhängkeitsliste
`$^`	Gesamte Abhängkeitsliste
`$?`	Liste aller Abhängigkeiten, die neuer als das Target sind
`$(@D)`	Verzeichnisname von `$@`
`$(@F)`	Reiner Dateiname (ohne Verzeichnisname) von `$@`
`$(CURDIR)`	Name des Verzeichnisses, in dem `make` gerade arbeitet

Beispiel: $@

```
$ cat Makefile
.SUFFIXES: bar
foo.bar: foo bar
    @echo $@
$ make
foo.bar
```

Beispiel: $*

```
$ cat Makefile
.SUFFIXES: bar
foo.bar: foo bar
    @echo $*
$ make
foo
```

„Eingebaute" Makros

Makro	Bedeutung
`$@`	Voller Name des Target
`$*`	Name des Target ohne Dateiendung
`$<`	**Erste Abhängigkeit aus der Abhängkeitsliste**
`$^`	**Gesamte Abhängkeitsliste**
`$?`	Liste aller Abhängigkeiten, die neuer als das Target sind
`$(@D)`	Verzeichnisname von `$@`
`$(@F)`	Reiner Dateiname (ohne Verzeichnisname) von `$@`
`$(CURDIR)`	Name des Verzeichnisses, in dem `make` gerade arbeitet

Beispiel: $<

```
$ cat Makefile
foo.bar: foo bar
    @echo $<
$ make
foo
```

Beispiel: $^

```
$ cat Makefile
foo.bar: foo bar
    @echo $^
$ make
foo bar
```

„Eingebaute" Makros

Makro	Bedeutung
`$@`	Voller Name des Target
`$*`	Name des Target ohne Dateiendung
`$<`	Erste Abhängigkeit aus der Abhängkeitsliste
`$^`	Gesamte Abhängkeitsliste
`$?`	**Liste aller Abhängigkeiten, die neuer als das Target sind**
`$(@D)`	Verzeichnisname von `$@`
`$(@F)`	Reiner Dateiname (ohne Verzeichnisname) von `$@`
`$(CURDIR)`	Name des Verzeichnisses, in dem `make` gerade arbeitet

Beispiel: $? (1)

```
$ touch foo.bar
$ cat Makefile
foo.bar: foo bar
    @echo $?
$ touch bar
$ make
bar
```

Beispiel: $? (2)

```
$ touch foo.bar
$ cat Makefile
foo.bar: foo bar
    @echo $?
$ touch foo
$ make
foo
```

„Eingebaute" Makros

Hochschule RheinMain

Makro	Bedeutung
`$@`	Voller Name des Target
`$*`	Name des Target ohne Dateiendung
`$<`	Erste Abhängigkeit aus der Abhängkeitsliste
`$^`	Gesamte Abhängkeitsliste
`$?`	Liste aller Abhängigkeiten, die neuer als das Target sind
`$(@D)`	**Verzeichnisname von `$@`**
`$(@F)`	**Reiner Dateiname (ohne Verzeichnisname) von `$@`**
`$(CURDIR)`	**Name des Verzeichnisses, in dem `make` gerade arbeitet**

Beispiel: `$(@D)`, `$(@F)`

```
$ cat Makefile
../foo.bar: foo bar
        @echo $@
        @echo $(@F)
        @echo $(@D)
$ make
../foo.bar
foo.bar
..
```

Beispiel: `$(CURDIR)`

```
$ cat Makefile
foo.bar: foo bar
        @echo $(CURDIR)
        @echo `pwd`
$ make
/home/kaiser/make_test
/home/kaiser/make_test
```

Variablen: Grundsätzliches ...

- Im Makefile können Variablen vereinbart ...

```
MYVAR=foo bar
```

- ... und mit $(<name>) oder ${<name>} abgerufen werden:

```
all: $(MYVAR)
        @echo ${MYVAR} foobar
```

Ergebnis:

```
$ make
foo bar foobar
```

- Variablen können über die Kommandozeile überschrieben werden:

```
$ make MYVAR="blah blubb"
blah blubb foobar
```

... und Spezielles

Pattern-Substitution

```
FILES=foo.c bar.c
OBJS=$(FILES:.c=.o)
all:
        @echo $(FILES)
        @echo $(OBJS)
```

Ergebnis

```
$ make
foo.c bar.c
foo.o bar.o
```

Variablen erweitern

```
FILES=foo.c bar.c
OBJS=$(FILES:.c=.o)
FILES += foobar.c
all:
        @echo $(FILES)
        @echo $(OBJS)
```

Ergebnis

```
$ make
foo.c bar.c foobar.c
foo.o bar.o foobar.o
```

Shell-Variablen

```
OBJS=$(FILES:.c=.o)
FILES += foobar.c
all:
        @echo $(FILES)
        @echo $(OBJS)
```

Ergebnis

```
export FILES=blah.c
$ make
blah.c foobar.c
blah.o foobar.o
```

Eingebaute Variablen

- Make besitzt „eingebaute", vorbesetzte Variablen:

Name	Defaultwert	Bedeutung	Name	Defaultwert
CC	cc	C-Compiler	CFLAGS	-
CXX	g++	C++-Compiler	CXXFLAGS	-
CPP	$(CC) -E	C-Preprozessor	CPPFLAGS	-
LD	ld	Linker	LDFLAGS	-
AR	ar	Archiver	ARFLAGS	rv
AS	as	Assembler	ASFLAGS	-
RM	rm -f	Remove	-	-

- Bezeichnen Tools und zugehörige default-Flags
- Können über Kommandozeile oder Shell-Environment überschrieben werden:

```
$ make CC=mips-elf-gcc CFLAGS=-Wall
```

Regelmuster

- Vielfach soll die gleiche Regel auf mehrere Dateien angewendet werden

→ Muster-Regeln: 2 Formen:

1. Form: Implizite Regel

```
.SUFFIXES:
.SUFFIXES: .c .o
.c.o:
        $(CC) $(CFLAGS) $<
```

- `.SUFFIXES: .c .o` erklärt Dateiendungen (.c und .o sind normalerweise bereits „eingebaut")
- `.SUFFIXES:` löscht bereits bestehende Dateiendungen

Regelmuster

- Vielfach soll die gleiche Regel auf mehrere Dateien angewendet werden

→ Muster-Regeln: 2 Formen:

2. Form: Muster-Regel

```
%.o: %.c
        $(CC) $(CFLAGS) $<
```

- i.W. gleiche Funktionalität.
- Implizite Regeln (1.Form) gelten als „deprecated"

Eingebaute Regelmuster

- Make besitzt ca. 90 „built-in rules", wie z.B. (Zitat):

```
COMPILE.c = $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c
OUTPUT_OPTION = -o $@
%.o: %.c
        $(COMPILE.c) $(OUTPUT_OPTION) $<
```

- Befehl zum Anzeigen der gesamten Regelbasis:

```
make -p
```

- Damit können Makefiles extrem kurz werden

Beispiel: Mikro-Makefile

```
hello:
```

→ Erzeugt Programm `hello` aus `hello.c`
(..oder aus `hello.cpp`, `hello.o`, `hello.s`...)

- Optionen können von der Kommandozeile „eingeschleust" werden

```
$ make CFLAGS="-g -DNDEBUG"
```