Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2014

Herausforderung: Schreiben portabler Software

Damit ein Projekt auf verschiedenen Unix-Systemen übersetzt werden kann, sind verschiedene Schritte nötig:

- Prüfen auf
  - Existenz des C-Compilers und weiterer notwendiger Programme,
  - Existenz benötigter Header-Dateien
- Anpassen des Makefiles an die Begebenheiten des Systems:
  - Name des C-Compilers und sonstiger Programme
  - Installationspfade

Autotools

Für die automatische Konfiguration des Makefiles an das lokale System existiert das Paket *autotools*, bestehend aus den Programmen *autoheader* und *autoconf*. Hierbei entstehen aus einem generischen Makefile (`Makefile.in`, das wiederum mit *automake* generiert werden kann) und den vorhandenen Quellen ein Skript `configure`, das das Makefile erzeugt.

Typischer Ablauf:

```
./configure --prefix=/usr
make all
make install
```


Beispiel: ringbuffer

Als Beispiel soll aus den Quellen vom *ringbuffer* aus dem ersten Teil der Vorlesung eine dynamische Bibliothek generiert werden, die dann ebenso wie die Header-Datei und die Man-Page installiert wird.
Hierzu wird folgendes `Makefile.in` benutzt:

```
top_srcdir = @top_srcdir@
srcdir = @srcdir@

CC = @CC@
CFLAGS = -I$(top_srcdir) -I$(srcdir) @CFLAGS@
LDFLAGS = @LDFLAGS@
LIBS = @LIBS@
INSTALL = @INSTALL@
INSTALL_DATA = @INSTALL_DATA@
LN_S = @LN_S@
```


Makefile.in (Forts.)

```
prefix = @prefix@
exec_prefix = @exec_prefix@
libdir = @libdir@
includedir = @includedir@
datarootdir = @datarootdir@
datadir = @datadir@
mandir = @mandir@

.PHONY: all install clean

SOVERSION = 1
RELEASE = 0

SOURCES = ringbuffer.c
HEADERS = ringbuffer.h
man3dir = $(mandir)/man3
man3_MANS = ringbuffer.3
```

Makefile.in (Forts.)

```
all: libringbuffer.so.$(SOVERSION).$(RELEASE)

libringbuffer.so.$(SOVERSION).$(RELEASE): $(SOURCES:.c=.o)
    $(CC) -shared -o $@ $(LDFLAGS) \
          -Wl,-soname=libringbuffer.so.$(SOVERSION) $+ $(LIBS)

install:
    $(top_srcdir)/mkinstalldirs $(DESTDIR)$(libdir)
    $(top_srcdir)/mkinstalldirs $(DESTDIR)$(includedir)
    $(INSTALL) libringbuffer.so.$(SOVERSION).$(RELEASE) \
               $(DESTDIR)$(libdir)
    $(INSTALL_DATA) $(HEADERS) $(DESTDIR)$(includedir)
    (cd $(DESTDIR)$(libdir); \
    rm -f libringbuffer.so.$(SOVERSION); \
    $(LN_S) libringbuffer.so.$(SOVERSION).$(RELEASE) \
            libringbuffer.so.$(SOVERSION); \
    rm -f libringbuffer.so; \
    $(LN_S) libringbuffer.so.$(SOVERSION) libringbuffer.so)
    $(top_srcdir)/mkinstalldirs $(DESTDIR)$(man3dir)
    for man in $(man3_MANS); do \
        $(INSTALL_DATA) $$man $(DESTDIR)$(man3dir); \
    done
```


Makefile.in (Forts.)

```
clean:
    rm -f libringbuffer.so.$(SOVERSION).$(RELEASE) \
          $(SOURCES:.c=.o)

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

%.d: %.c
    $(CC) -MM $(CFLAGS) $< | sed 's,\($*\).o[ ]*:,\1.o $@:,' > $@

ifneq ($(MAKECMDGOALS),clean)
include $(SOURCES:.c=.d)
endif
```


Generieren des configure-Skripts

In diesem einfachen Beispiel wird folgende Datei `configure.ac` benutzt:

```
#                                               -*- Autoconf -*-
# Process this file with autoconf to produce a configure script.

AC_PREREQ([2.68])
AC_INIT([ringbuffer], [1.0], [mail@example.com])
AC_CONFIG_SRCDIR([ringbuffer.h])

# Checks for programs.
AC_PROG_CC
AC_PROG_INSTALL
AC_PROG_LN_S

AC_CONFIG_FILES([Makefile])
AC_OUTPUT
```

Hieraus wird dann mit `autoconf` das Skript `configure` erzeugt, mit dem dann das richtige Makefile automatisch generiert werden kann.

Ausblick: Prüfen auf verschiedene Header-Files

Die Stärke der *autotools* liegt darin, dass Quelldateien analysiert werden können, um Tests auf verschiedene Header-Files, Funktionen etc. zu generieren:

```
#ifdef HAVE_CONFIG_H
# include <config.h>
#endif

#include <stdio.h>
#ifdef STDC_HEADERS
# include <stdlib.h>
# include <stddef.h>
#else
# ifdef HAVE_STDLIB_H
# include <stdlib.h>
# endif
#endif
#ifdef HAVE_STRING_H
# if !defined STDC_HEADERS && defined HAVE_MEMORY_H
#  include <memory.h>
# endif
# include <string.h>
#endif
```


Workflow der *autotools*

- `autoscan` analysiert den Quelltext und erzeugt `configure.scan` mit folgendem Inhalt:

  ```
  AC_CONFIG_HEADERS([config.h])
  AC_CHECK_HEADERS([memory.h stddef.h stdlib.h string.h])
  ```

- Nach Erzeugen von `configure.ac` anhand von `configure.scan` wird mit `autoheader` die Datei `config.h.in` generiert.
- Mit `autoconf` wird das Skript `configure` erzeugt.
- Durch Aufruf von `configure` entsteht das `config.h` mit den Informationen darüber, welche der verlangten Headerdateien auf dem System vorhanden sind.

RPMs: Grundlagen

Die meisten Linux-Distributionen benutzen das Format des *Redhat Package Managers* für die Installation und Deinstallation von Softwarepaketen (Alternative: *APT – Advanced Packaging Tool auf Debian & Co.*).

Was leistet RPM:

- Prüfen auf Abhängigkeiten/Konflikten mit anderen Paketen vor der Installation (z. B. wird die grafische CD-Brennoberfläche K3b nur installiert, wenn auch die eigentlichen (Kommandozeilen-)Brennprogramme installiert sind),
- Installation und Deinstallation von Paketen (mit Vor- und Nacharbeiten, z. B. Ausführen von `ldconfig`),
- Nachverfolgen, welche Dateien zu welchem Software-Paket gehören, Prüfung auf Integrität,
- Update von Paketen

Das Programm rpm

Auf der Kommandozeile wird mit dem Befehl `rpm` gearbeitet:

`rpm -i dateiname.rpm` zum Installieren eines RPM (falls die Abhängigkeiten erfüllt sind).

`rpm -e paket` zum Deinstallieren eines Paketes.

`rpm -U dateiname.rpm` installiert die angegebene RPM-Datei, ältere Versionen (so vorhanden) werden deinstalliert.

`rpm -q paket` zeigt an, ob und in welcher Version das Paket installiert ist.
`rpm -qi paket` gibt Informationen zum Paket aus.

`rpm -ql paket`, `rpm -qd paket`, `rpm -qc paket`
Liste aller zum Paket gehörenden Dateien bzw. Liste der Dokumentations- und Konfigurationsdateien.
`rpm -V paket` zum Prüfen der Integrität des Pakets.

Bauen von RPMs

Zum Erstellen von RPMs für ein Projekt benötigt man

- die Quellen des Projekts
- evtl. Patches für die Anpassung an die Distribution/Architektur
- eine *SPEC*-Datei mit den Anweisungen, wie der RPM erzeugt werden soll.

Der RPM-Build erfordert eine bestimmte Dateistruktur:

```
topdir/SPECS
      /SOURCES
      /BUILD
      /RPMS
      /SRPMS
```

wobei das Toplevelverzeichnis in `~/.rpmmacros` definiert wird:

```
%_topdir /home/klaus/rpms
```


Einfache SPEC-Datei

```
Summary: Demon to ping syslog messages
Name: logdaemon
Version: 1.0
Release: 1
License: Public Domain
Group: System/Services
Prefix: /usr
Source: logdaemon-1.0.tar.gz
Packager: Santa Claus <sclaus@northpole.com>
BuildRoot: /tmp/logdaemon-1.0-root

%description
This is just a test RPM

%prep
%setup
```

Einfache SPEC-Datei (Forts.)

```
%build
./configure --prefix=%{_prefix} --sysconfdir=%{_sysconfdir}
make all

%install
make install DESTDIR=${RPM_BUILD_ROOT}

%files
%doc README
%{_sbindir}/log
%{_initrddir}/log
```


Anatomie der SPEC-Datei

Wie man sieht, besteht die SPEC-Datei aus verschiedenen Abschnitten:

- `%prep` Auspacken der Quellen, evtl. Anwenden von Patches
- `%build` Kompilieren des Projektes
- `%install` Installieren der Dateien (unterhalb von *BuildRoot*)
- `%files` Dateien, die in den RPM übernommen werden sollen (ggfs. mit Markierung als Dokumentation oder Konfigurationsdatei)

Dabei kann eine SPEC mehrere RPM-Pakete definieren, in die die Dateien aufgesplittet werden (z. B. Trennung Programm und Headerdateien). Befehle vor oder nach der (De-)Installation können in den Abschnitten `%pre`, `%post` bzw. `%preun`, `%postun` definiert werden.

Aufruf von `rpmbuild`

Der oder die RPMs werden dann mit `rpmbuild` erzeugt:

- `-bb` Erzeugen des (binary) RPMs,
- `-bs` erzeugen des Source-RPMs (SRPM),
- `-ba` erzeugen von allen beiden.
- `-bp` Nach `%prep` aufhören,
- `-bc` nach `%build` aufhören,
- `-bi` nach `%install` aufhören.

`--short-circuit` sorgt in Verbindung mit `-bc` bzw. `-bi` dafür, dass die Schritte *vorher* übersprungen werden.

Grundlagen Netzwerk

Für Netzwerke wird heutzutage meist noch IPv4 benutzt, bei der einer Netzwerkschnittstelle eine IP-Adresse zugeordnet wird, die aus vier Bytes besteht. Der Übergang zu IPv6 findet derzeit statt, aufgrund des allmählich knapp werdenden Adress-Pools aber überraschend langsam.

Hierbei gibt es einige besondere Adressbereiche:

127.0.0.0/8 bezieht sich auf das Loopback-Interface. Dies ist keine reale Netzwerkkarte, sondern eine Rechner-interne Kommunikation.

10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16 sind private IP-Bereiche, insbesondere für Intranets. Private IP-Adressen werden nicht innerhalb des öffentlichen Internets geroutet und sind daher unsichtbar. Die Verbindung zum Internet erfolgt über Proxys bzw. per NAT (*Network Address Translation*).

Subnetze und Gateways

Neben der IP-Adresse ist für eine Netzwerkschnittstelle noch bedeutsam, über welchen IP-Bereich sich das Subnetz erstreckt, in dem andere Netzwerkendpunkte direkt erreicht werden. Dies sind Rechner, bei denen ein Bit-And von IP mit Netmask zum selben Wert führen.

Beispiele:

- IP 192.168.129.27, Netmask 255.255.255.0 (Subnet 192.168.129.0/24): Rechner im IP-Bereich 192.168.129.0–255 sind im selben Subnet.
- IP 10.10.103.123, Netmask 255.255.240.0 (Subnet 10.10.96.0/20): Rechner im IP-Bereich 10.10.96–111.0–255 sind im selben Subnet.

Konfiguration der Netzwerkschnittstelle per Hand

- Konfiguration des Loopback-Interfaces:
  ```
  ip addr add 127.0.0.1/8 dev lo
  ```
  oder
  ```
  ifconfig lo 127.0.0.1 netmask 255.0.0.0
  ```
- Konfiguration der Netzwerkkarte
  ```
  ip addr add 10.10.103.123/20
  broadcast 10.10.111.255 dev eth0
  ```
  oder
  ```
  ifconfig eth0 10.10.103.123
  netmask 255.255.240.0 broadcast 10.10.111.255
  ```

Hinzufügen der Routen

Nun müssen noch die Routen hinzugefügt werden:

- Verbindungen nach 127.0.0-1/8 über das Loopback-Interface
- Verbindungen nach 10.10.96.0/20 direkt über eth0
- Verbindungen nach außerhalb über das Gateway 10.10.96.1

```
route add -net 127.0.0.0 netmask 255.0.0.0 dev lo
route add -net 10.10.96.0 netmask 255.255.240.0 dev eth0
route add default gw 10.10.96.1
```