

Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2011/2012

1/22

popen

Programmentwicklung unter Linux

make

gcc, objdump, readelf

2/22

Die Funktion popen

Im letzten Beispiel wurden `fork`, Pipes und die `exec`-Funktionen miteinander verknüpft:

- Per `fork` wurde ein Kindprozess erzeugt,
- dieser benutzte eine vorher angelegte Pipe, wobei `stdin` auf die lesende Seite der Pipe umgebogen wurde.
- Dann wurde der Kindprozess durch eine `bash` ersetzt, die über die Pipe aus dem Elternprozess Befehle bekam.

Letztlich geht diese Aktion allerdings einfacher, nämlich mit

```
#include <stdio.h>
```

```
FILE* popen(const char* cmd, const char* type)
```

wobei `type` entweder „r“ oder „w“ ist.

3/22

Die Funktion popen (Forts.)

Die Pipe an das ausgeführte Programm wird mit `pclose(FILE *stream)` geschlossen. Hierbei wird (mit `waitpid`) auf das Ende des aufgerufenen Programms gewartet, es wird also erwartet, dass dieses bei EOF auf `stdin` bzw. beim Schließen von `stdout` endet.

Die Kommunikation mit dem externen Programm ist unidirektional, also können *entweder* Eingaben an das Programm geschickt *oder* dessen Ausgabe gelesen werden. Es gibt in der C-Bibliothek kein `popen2` mit bidirektionaler Kommunikation, dieses muss entweder selbst implementiert oder von dritter Seite bereit gestellt werden.

4/22

Beispiel

Im folgenden Beispiel soll aus einem Programm heraus eine E-Mail verschickt werden (wie es z. B. Backup-Programme bei Fehlern bei der Datensicherung tun). Hierfür wird das Programm `sendmail` aufgerufen, das unter Linux Mails entgegennimmt und transportiert.

```
#include <stdio.h>

int main() {
    FILE* p = popen("/usr/lib/sendmail -t -oem -oi", "w");
    fprintf(p, "From: klaus\n");
    fprintf(p, "To: karl\n");
    fprintf(p, "Subject: Test\n");
    fprintf(p, "\n");
    fprintf(p, "Hallo\n");
    pclose(p);
    return(0);
}
```

5/22

Beispiel zur Motivation

Normalerweise bestehen C-Programme aus mehreren Quelltext- (`.c`) und Header-Dateien (`.h`), die einzeln *kompiliert* und dann zu einem ausführbaren Programm *gelenkt* werden.

Datei `main.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <myfunc.h>

int main() {
    int i = myfunc(3);
    printf("Die Antwort heisst: %d\n", i);
    return(EXIT_SUCCESS);
}
```

6/22

popen Programmentwicklung unter Linux make gcc, objdump, readelf

Beispiel (Forts.)

Datei `myfunc.h`:

```
#ifndef _MYFUNC_H
#define _MYFUNC_H

int myfunc(int);

#endif
```

Datei `myfunc.c`:

```
#include <myfunc.h>

int myfunc(int x) {
    return(x*2);
}
```

7/22

popen Programmentwicklung unter Linux make gcc, objdump, readelf

Editoren/IDEs

Prinzipiell können C-Programme mit jedem beliebigen Texteditor geschrieben werden, z. B. *vi*, *emacs*, *keddit*, *nano*. Teilweise bieten diese bereits gewisse Unterstützung bei der Programmentwicklung. z. B. Syntax-Highlighting.

Speziell für die Programmentwicklung existieren spezielle Editoren bzw. Entwicklungsumgebungen (IDE – *integrated development environment*):

- Eclipse mit CDT-Plugin
- KDevelop
- Bluefish
- Geany

8/22

popen Programmentwicklung unter Linux make gcc, objdump, readelf

Einführung: Manuelles Kompilieren und Linken

Aus den einzelnen Quelltexten wird das Programm mit folgenden Befehlen kompiliert und gelinkt:

```
gcc -c -I. -Wall main.c
gcc -c -I. -Wall myfunc.c
gcc -o prog main.o myfunc.o
```

Während des Entwicklungsprozesses muss auf folgende Dinge geachtet werden:

- Ändert sich eine der beiden C-Dateien, so muss diese neu kompiliert, also eine neue Object-Datei erzeugt werden.
- Hat sich eine der beiden Object-Dateien geändert, so muss neu gelinkt werden.

9/22

Grundlagen von Makefiles

Der Workflow zum Kompilieren und Linken eines Programms aus verschiedenen Dateien lässt sich automatisieren.

Betrachten wir folgende Datei mit dem Namen **Makefile**:

```
CC=gcc
CFLAGS=-I. -Wall

prog: main.o myfunc.o
    $(CC) -o $@ $+

myfunc.o: myfunc.c myfunc.h
    $(CC) -c $(CFLAGS) $<

main.o: main.c myfunc.h
    $(CC) -c $(CFLAGS) $<
```

10/22

Regeln in Makefiles

Eine Regel in einem Makefile hat folgende Form:

```
target: dep1 dep2 ...
(TAB) Anweisung1
(TAB) Anweisung2
(TAB) ...
```

wobei innerhalb der Anweisungen folgende Variablen definiert sind:

- `$@` Name des Targets,
- `$+` Alle Dateien, von denen das Target abhängt,
- `$<` Erste Datei, von der das Target abhängt.

11/22

Workflow im Beispiel

- Beim Befehl `make prog` soll das gleichnamige Ziel erzeugt werden. Im Makefile steht, dass das Programm von `main.o` und `myfunc.o` abhängt.
- `main.o` und `myfunc.o` hängen jeweils von der entsprechenden C-Datei ab sowie beide von `myfunc.h` ab.
- Abhängigkeiten werden rekursiv geprüft:
 - Ist eine der Object-Dateien älter als die entsprechende C-Datei oder `myfunc.h`, so wird die entsprechende C-Datei neu kompiliert.
 - Ist ein evtl. schon vorhandenes Programm `prog` älter als eine der beiden Object-Dateien, so wird neu gelinkt.

12/22

Phony Targets

Häufig finden sich in Makefiles Targets wie:

```
clean:
    rm -f abc.o xyz.o
```

Hier wird im Ggs. zu normalen Targets keine Datei erzeugt (sondern in diesem Fall beim Build-Prozess erzeugte Dateien gelöscht). Trotzdem funktioniert das Target, es sei denn, es existiert eine Datei, die zufällig „clean“ heißt. Dann erscheint beim Aufruf von `make clean` die Meldung, dass „clean“ schon aktuell sei.

Um dies zu verhindern, können Targets, die keine Dateien bezeichnen, als *phony* deklariert werden:

```
.PHONY: clean
```

```
clean:
    rm -f abc.o xyz.o
```

13/22

Implizite Regeln

Die Regeln zum Kompilieren von `main.c` und `myfunc.c` im Beispiel vom Anfang sind faktisch identisch. Diese lassen sich durch eine allgemeine Vorschrift zum Kompilieren ersetzen.

Allerdings muss hier dann noch explizit die Abhängigkeit der Object-Dateien von `myfunc.h` angegeben werden, sonst wird bei Änderungen in dieser Datei nicht mehr automatisch kompiliert.

```
myfunc.o: myfunc.h
main.o: myfunc.h
```

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

14/22

Inkludieren von Makefiles

In einem Makefile können weitere Makefiles importiert werden, z. B.:

```
include filename.mk
```

Dies kann verschiedene Funktionen haben:

- Bei einem großen Projekt mit mehreren Makefiles wird z. B. die Datei „global.mk“ inkludiert, die in allen Makefiles benutzte Definitionen enthält;
- es werden Dateien inkludiert, die Abhängigkeitsinformationen für Quelldateien enthalten.

Hierbei können inkludierte Makefiles von GNU-Make automatisch regeneriert werden, wenn diese selber Target sind. Dies ist insbesondere im zweiten Fall sehr nützlich, damit die Abhängigkeiten der Quelldateien immer aktuell sind.

15/22

Automatisches Bestimmen von Abhängigkeiten

Der GNU-C-Compiler hat die Optionen `-M` und `-MM`, die einen Make-Schnipsel mit den Abhängigkeiten einer C-Datei ausgeben (im ersten Fall inklusive, im zweiten ohne System-Header).

So kann der Befehl `gcc -MM main.c` folgende Ausgabe erzeugen:

```
main.o: main.c myfunc.h
```

Dies kann man sich mit folgenden Zeilen im Makefile zunutze machen:

```
include main.d
```

```
%.d:%.c
    gcc -M $< | sed 's,\($*\)\.o[ ]*:, \1.o $@ :,g' > $@
```

16/22

Konvention: Unterstützen von DESTDIR

Betrachten wir folgendes Makefile:

```
INSTALL_EXE=install -m755 -D
```

```
install: $(DESTDIR)/usr/bin/myprog
```

```
$(DESTDIR)/usr/bin/myprog: myprog
    $(INSTALL_EXE) $< $@
```

Mit `make install` wird „myprog“ wie gewünscht nach `/usr/bin` installiert.

Sollen die Dateien des Projekts in einem anderen Verzeichnisbaum installiert werden, so ruft man auf:

```
make install DESTDIR=/tmp/myprog
```

Dies ist hilfreich beim Erstellen von Software-Releases, die als Archiv weitergegeben und auf einem anderen Rechner installiert werden sollen.

17/22

gcc: Format der Ausgabedatei

Normalerweise versucht der GNU-C-Compiler `gcc` direkt ein ausführbares Programm zu erzeugen, beim Aufruf von `gcc xyz.c` unter dem Namen `a.out`, bei `gcc -o myprog xyz.c` als `myprog`.

Mit folgenden Optionen kann die Verarbeitung der Datei vorher abgebrochen werden:

- E Nur Präprozessing, z. B. Ersetzen von `#define`.
- S Nur Kompilieren, ohne Assemblieren. Ausgabe ist eine Datei mit Assemblercode unter dem Namen `xyz.s`.
- c Kompilieren und Assemblieren, aber nicht Linken. Ausgabedatei heißt `xyz.o`.

18/22

Weitere Optionen

- Wall Zeigt besonders viele Warnungen an.
- Werror Warnungen werden als Fehler behandelt, führen also zum Abbruch.
- I Zusätzliches Verzeichnis, wo Headerdateien gesucht werden.
- l Eine Bibliothek wird dazugelinkt. Achtung: vor den Namen wird automatisch der String „lib“ gestellt, bei `-lpthread` wird also die Datei „libpthread.so“ (dynamische Bibliothek) bzw. „libpthread.a“ (statisch) hinzugelinkt.
- L Zusätzliches Verzeichnis, wo Bibliotheken gesucht werden.

19/22

Was passiert beim Kompilieren eines Quelltextes?

Betrachten wir folgenden Quelltext in der Datei „test.c“:

```
#include <stdio.h>

int glob_var1 = 7;
int glob_var2;
const int glob_var3 = 99;
int glob_var4 = 77;

int main() {
    int lokal_var = 111;
    int glob_var2 = 192;
    printf("Werte: %d %d %d %d\n",
        glob_var1, glob_var2, glob_var3, lokal_var);
    return(0);
}
```

20/22

Sections für Anweisungen und Daten

Beim Kompilieren landen die globalen Variablen in folgenden *Sections*:

- Initialisierte Daten in der Section `.data` (les- und schreibbar)
- Konstante Daten in der Section `.rodata` (nur lesbar)
- Nicht initialisierte globale Variablen werden wie lokale Variablen auf dem Stack abgelegt.
- Der Maschinencode der Funktion `main()` landet in der Section `.text`.

Diese Sections lassen sich über der Prozess zum Erzeugen des ausführbaren Programms mit den Programmen `objdump` und `readelf` nachverfolgen.

21/22

Optionen

Optionen von `objdump`:

- h Vorhandene Sections
- t Tabelle der verwendeten Symbole
- d -j `.text` Disassemblieren der Section `.text`
- s -j `.data` Inhalt der Section `.data`

Optionen von `readelf`

- h Header-Informationen zum Programm
- S Sections
- s Symbole
- x `.rodata` Hexdump der Section `.rodata`