Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2011/2012

Libraries

Signale

Bibliotheken

Bisher wurden jeweils alle Quellen einzeln kompiliert und dann zusammengelinkt.

Sollen kompilierte C-Quellen häufig verwendet werden, so können diese zu Bibliotheken zusammen geschnürt werden, die beim Erzeugen eines Programms hinzugelinkt werden.

Hierbei unterscheidet man zwei Fälle:

- Statische Bibliothek, deren Inhalt tatsächlich *beim Linken* in das ausführbare Programm geschrieben wird (tatsächlich ist das Programm also hinterher genauso groß, als wäre es aus den Einzelquellen gelinkt worden).
- Dynamische Bibliotheken, bei denen der Code der hinzugelinkten Funktionen erst *zur Laufzeit* dynamsich dazugelinkt wird. Hierdurch wird das Programm kleiner, aber die dynamische Bibliothek muss nun natürlich auch zur Laufzeit gefunden werden!

Statische Bibliotheken

Bei einer statischen Bibliothek wird das Programm `ar` verwendet:

```
ar -r libmytest.a abc.o xyz.o
```

(mit der Option `-t` wird eine Tabelle der enthaltenen Objectdateien in einer Bibliothekt angzeigt, mit `-d` wird eine Objectdatei entfernt).
Anschließend wird mit `ranlib libmytest.a` die Bibliothekt indiziert.

Nun kann die Bibliothek hinzugelinkt werden:

```
gcc -o prog prog.o -L. -lmytest
```

Dynamische Bibliotheken

Nun wird eine dynamische Bibliothek erzeugt:
`gcc -shared -o libmytest.so abc.o xyz.o`
Nach `gcc -o prog prog.o -L. -lmytest` ist das Programm nun dynamisch gegen die Bibliothek gelinkt.

Mit `ldd prog` wird angezeigt, gegen welche dynamischen Bibliotheken das Programm gelinkt ist – und ob bzw. wo diese gefunden werden.
Hierbei wird `libmytest.so` i. A. erstmal *nicht* gefunden! Dies ist erst der Fall, nachdem die Variable `LD_LIBRARY_PATH` so gesetzt wurde, dass das Verzeichnis mit der Bibliothek enthalten ist, z. B.
`export LD_LIBRARY_PATH=.`

Der soname

Dynamisches Linken hat ein Problem: ändert sich die Bibliothek, so sind alte Programme gegen eine wahrscheinlich inkompatible Bibliothek gelinkt.

Daher werden dyn. Bibliotheken mit Versionsnummern versehen:

```
gcc -shared -o libmytest.so.1.0 -Wl,-soname=libmytest.so.1 abc.o xyz.o
ln -s libmytest.so.1.0 libmytest.so.1
ln -s libmytest.so.1 libmytest.so
```

Nun wird *beim Linken* zwar wie bisher `libmytest.so` genommen, aber als Linkinformation der SO-Name verwendet, also `libmytest.so.1`, wie man auch mit `ldd` prüfen kann.
Physikalisch heißt die Bibliothek `libmytest.so.1.0`.

Ablauf bei Änderungen

1. Kompatible Änderung: Es wird eine neue Bibliothek `libmytest.so.1.1` erzeugt, aber der SO-Name beibehalten und der Link `libmytest.so.1` auf die neue Datei umgelegt. Alte wie neue Programme verwenden die neue Bibliothek.
2. Inkompatible Änderung: Es wird `libmytest.so.2.0` mit dem SO-Namen `libmytest.so.2` erzeugt. Der Link `libmytest.so` zeigt auf den neuen SO-Namen. Neue Programme werden gegen die Version 2 gelinkt, alte verwenden wie bisher Version 1 und funktionieren daher immer noch.

ldconfig

Es ist lästig, wenn zum Funktionieren eines Programms `LD_LIBRARY_PATH` gesetzt sein muss.

Für das automatische Finden von dyn. Bibliotheken gibt es im System einen Cache mit Namen und Speicherorten. Damit die dyn. Bibliotheken in einem Verzeichnis im Cache auftauchen, müssen folgende Aktionen erfolgen:

1. Der Name des Verzeichnisses befindet sich in einer Textdatei `/etc/ld.so.conf.d/*.conf` (früher direkt in der Datei `/etc/ld.so.conf`),
2. der Cache wird mit `ldconfig` aktualisiert.

Mit `ldconfig -p` wird der Inhalt des Caches aufgelistet, der Befehl `ldconfig -p | grep libmytest.so` ergibt also, ob diese nun im Cache ist.

Motivation für Signale

Betrachten Sie folgendes Programm:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>

int main() {
    FILE* out = fopen("zahlen.txt","w");
    srand(time(NULL));
    while (1) {
        fprintf(out,"%d\n",rand());
        sleep(1);
    }
}
```

Dieses Programm (samt Endlosschleife) kann mit Ctrl-C unterbrochen werden, nur ist die Datei dann leer, weil sie nicht geschlossen wird.

Signale

Beim Ctrl-C innerhalb der Bash wird an den gerade ausgeführten Prozess das Signal 2 (SIGINT) geschickt. Dieses ist eines von vielen Signalen:

Signal	Nr.	Bedeutung
SIGHUP	1	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Interrupt from keyboard
SIGQUIT	3	Quit from keyboard
SIGILL	4	Illegal Instruction
SIGABRT	6	Abort signal from abort(3)
SIGFPE	8	Floating point exception
SIGKILL	9	Kill signal

Signale (Forts.)

Signal	Nr.	Bedeutung
SIGSEGV	11	Invalid memory reference
SIGPIPE	13	Broken pipe: write to pipe with no readers
SIGALRM	14	Timer signal from alarm(2)
SIGTERM	15	Termination signal
SIGUSR1	10	User-defined signal 1
SIGUSR2	12	User-defined signal 2
SIGCHLD	17	Child stopped or terminated
SIGSTOP	19	Stop process
SIGTSTP	20	Stop typed at tty

Behandlung von Signalen

Beim Auftreten eines Signals in einem Prozess wird der dafür definierte Signalhandler aufgerufen.

Dieser ist eine Funktion mit der Signalnummer als Integer-Parmeter und ohne Rückgabewert (also `void`.

Hierbei existieren zwei vordefinierte Signalhandler:

- SIG_DFL als der vom System definierte Default-Handler für das Signal und
- SIG_IGN als Handler, der das Signal ignoriert.

Signalhandler können (außer für `SIGKILL` und `SIGSTOP`!) selbst definiert werden.

Die Funktion sigaction

Zur Abfrage und Manipulation von Signalhandlern dient folgende Funktion:

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

Hierbei ist `signum` die Nummer des Signals, `act` und `oldact` Zeiger auf eine Struktur vom Typ `sigaction`, mit deren Daten der Signalhandler neu gesetzt wird bzw. in der anschließend die Konfiguration der Signalbehandlung vor dem Funktionsaufruf steht.

Wird für `act` ein Nullpointer übergeben, erfolgt keinerlei Änderung, in der von `oldact` referenzierten Struktur steht also die bestehende Konfiguration. Analog kann für `oldact` der Nullpointer übergeben werden, wenn bei einer Neukonfiguration die alte Konfiguration nicht interessiert.

Beispiel

```
#include <stdio.h>
#include <signal.h>
#include <string.h>
#include <errno.h>

#define MAX 70

void hdl(int signum) {
    printf("Caught signal %d\n", signum);
    fflush(stdout);
}

int main() {
    int signum;
    struct sigaction action;
    action.sa_flags = 0;
    sigemptyset(&action.sa_mask);
    action.sa_handler = SIG_IGN;
    sigaction(SIGUSR1, &action, NULL);
```

Beispiel (Forts.)

```
    action.sa_handler = hdl;
    sigaction(SIGUSR2, &action, NULL);
    for (signum=1; signum<=MAX; signum++) {
        if (sigaction(signum, NULL, &action)==0) {
            printf("Signal %d: %s, ", signum, strsignal(signum));
            if (action.sa_handler==SIG_DFL)
                printf("handled by default handler.");
            else if (action.sa_handler==SIG_IGN)
                printf("ignored.");
            else
                printf("handled by user defined handler");
            printf("\n");
        } else {
            if (errno==EINVAL)
                printf("Signal %d does not exist\n", signum);
        }
    }
    return(0);
}
```

Zurück zum Motivationsbeispiel

Nun kann im Beispiel aus der Motivation das Signal `SIGINT` so behandelt werden, dass die gerade geschriebene Datei korrekt geschlossen wird:

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>

FILE* out = NULL;

void hdl(int signum) {
    if (out) fclose(out);
    exit(0);
}
```

Zurück zum Motivationsbeispiel (Forts.)

```
void sethandler(int signum, void (*handler)(int)) {
    struct sigaction action;
    action.sa_handler = handler;
    action.sa_flags = 0;
    sigemptyset(&action.sa_mask);
    sigaction(signum, &action, NULL);
}

int main() {
    sethandler(SIGINT, hdl);
    out = fopen("zahlen.txt","w");
    srand(time(NULL));
    while (1) {
        fprintf(out,"%d\n",rand());
        sleep(1);
    }
}
```

Details von sigaction

Betrachten wir folgenden Quelltext:

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

void hdl(int signum) {
    printf("Erhaltenes Signal: %d\n", signum);
    sleep(10);
    printf("Signalbehandlung fertig\n");
}

int main() {
    struct sigaction action;
    sigemptyset(&action.sa_mask);
    action.sa_handler = hdl;
    action.sa_flags = 0;
    sigaction(SIGHUP,&action,NULL);
    sigaction(SIGUSR1,&action,NULL);
    while(1) {}
    return(0);
}
```

Details von `sigaction` (Forts.)

In diesem Beispiel haben nun `SIGHUP` und `SIGUSR1` denselben User-definierten Signalhandler, dessen Abarbeitung 10 Sekunden dauert.

Kommt innerhalb dieser 10 Sek. dasselbe Signal, so wird das zweite solange blockiert, bis die Abarbeitung des ersten Signals beendet ist. Anders sieht es aus, wenn während der Bearbeitungszeit des Signalhandlers für das erste Signal ein *anderes* Signal auftritt. Dann wird direkt in dessen Signalhandler gesprungen.

Dieses Verhalten kann geändert werden, indem in dem Element `sa_mask` der Struktur `sigaction` die Signale gesetzt werden, die während der Abarbeitung des Signalhandlers blockiert werden.

Blockieren mehrerer Signale

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#define MAX 32

void sethandlers(sigset_t *signals, void (*handler)(int)) {
    struct sigaction action;
    sigemptyset(&action.sa_mask);
    int signum;
    action.sa_mask = *signals;
    action.sa_flags = SA_RESTART;
    action.sa_handler = handler;
    for (signum=1; signum<=MAX; signum++) {
        if (sigismember(signals, signum))
            sigaction(signum, &action, NULL);
    }
}
```

Blockieren mehrerer Signale (Forts.)

```
void hdl(int signum) {
    printf("Erhaltenes Signal: %d\n", signum);
    sleep(10);
    printf("Signalbehandlung fertig\n");
}

int main() {
    sigset_t signals;
    sigaddset(&signals, SIGHUP);
    sigaddset(&signals, SIGUSR1);
    sethandlers(&signals, hdl);
    while(1) {}
    return(0);
}
```

Reagieren auf Ende eines Kindprozesses

Im folgenden Beispiel wird beim Ende eines Kindprozesses `wait` aufgerufen, damit kein Zombie übrig bleibt:

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

void hdl(int signum) {
    printf("Erhaltenes Signal %d: %s\n",
           signum, strsignal(signum));
    int status;
    int pid = wait(&status);
    printf("Beendeter Kindprozess: %d\n", pid);
}
```

Reagieren auf Ende eines Kindprozesses (Forts.)

```
int main() {
    int child = fork();
    if (child==0) {
        sleep(10);
        return(0);
    }
    struct sigaction action;
    sigemptyset(&action.sa_mask);
    sigaddset(&action.sa_mask, SIGCHLD);
    action.sa_handler = hdl;
    action.sa_flags = SA_RESTART;
    sigaction(SIGCHLD,&action,NULL);
    while (1) {}
    return(0);
}
```

Atomare Aktionen

Da ein Signal jederzeit auftreten kann, ist es möglicherweise notwendig, dass die Behandlung eines Signals verzögert wird, damit ein gewünschter Programmblock *atomar*, also ohne Unterbrechung abgearbeitet wird.

Für die folgenden Beispiele wird folgende Funktion als definiert vorausgesetzt:

```
void sethandler(int signum, void (*handler)(int)) {
    struct sigaction action;
    action.sa_handler = handler;
    action.sa_flags = SA_RESTART;
    sigemptyset(&action.sa_mask);
    sigaction(signum, &action, NULL);
}
```

Beispiel

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

int a, b;

void handler(int signum) {
    printf ("%d,%d\n", a, b);
    alarm(1);
}

int main() {
    sethandler(SIGALRM, handler);
    alarm (1);
    a = b = 0;
    while (1) {
        b = a = 1-a;
    }
}
```

Da die Anweisung `a=b=wert` nicht atomar ist, kann es sein, dass `a` und `b` bei der Ausgabe verschieden sind.

Atomare Variante des Beispiels

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

int a, b;
volatile sig_atomic_t flag = 0;

void handler(int signum) {
    flag = 1;
    alarm (1);
}

void myaction() {
    printf ("%d,%d\n", a, b);
    flag = 0;
}
```

Atomare Variante des Beispiels (Forts.)

```
int main() {
    sethandler(SIGALRM, handler);
    alarm (1);
    a = b = 0;
    while (1) {
        b = a = 1-a;
        if (flag) myaction();
    }
}
```

Nun wird durch den Signalhandler nur noch ein Flag gesetzt, die eigentliche Ausgabe erfolgt (bei gesetztem Flag) am Ende des Schleifenkörpers, wenn sicher gestellt ist, dass `a` und `b` gleich sind.

Funktionen zum Auslösen von Signalen

kill(pid, signum) schickt an den Prozess mit der angegebenen PID ein Signal.

raise(signum) schickt das Signal an den aktuellen Prozess, äquivalent zu `kill(getpid(),signum)`

abort() löst das Signal `SIGABRT` aus, das i. A. zum abrupten Programmende per Signal führt.

alarm(seconds) löst nach *seconds* Sekunden das Signal `SIGALRM` aus.