Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2014

Signale

Threads: Grundlagen

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-Parameter 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.

Threads unter Linux

Unter Linux werden unterschieden:

User Threads Verzahntes Ausführen von Programmsträngen innerhalb eines Prozesses, implementiert außerhalb des eigentlichen Betriebssystems in der `pthread`-Bibliothek (POSIX – Portable Operating System Interface), Scheduling durch Betriebssystem.

Kernelthreads laufen als Teil des Betriebssystems.

Unscharfe Unterscheidung: Multithreading – Multitasking

Im folgenden: Mehrere Stränge innerhalb einer Anwendung, also User Threads.

Erzeugen eines Threads

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

Hierbei wird ein neuer Thread angelegt, dessen Nummer in die Variable geschrieben wird, auf die der Zeiger `thread` zeigt (`pthread_t` ist äquivalent zu `unsigned long`).
In dem Thread wird dann die durch `start_routine` bezeichnete Funktion nebenläufig gestartet, an die `arg` übergeben wird.

`attr` enthält Einstellungen zum Thread, ist in der Praxis aber häufig `NULL`, wenn die Standardeinstellungen benutzt werden sollen.

Beispiel

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

void* run(void *arg) {
    int *ip = (int*) arg;
    printf("Thread %lu schlaeft %d s\n", pthread_self(), *ip);
    sleep(*ip);
    printf("Thread %lu fertig\n", pthread_self());
    pthread_exit(NULL);
}

int main() {
    pthread_t t1, t2;
    int dauer1 = 10;
    int dauer2 = 20;
    pthread_create(&t1, NULL, run, (void*) &dauer1);
    pthread_create(&t2, NULL, run, (void*) &dauer2);
```

Beispiel (Forts.)

```
    void *res;
    pthread_join(t1, &res);
    if (res==NULL) printf("Thread normal beendet\n");
    pthread_join(t2, NULL);
}
```

Mit `pthread_join` wird also auf das Endes des angegebenen Threads gewartet. Der Rückgabewert des Threads kann bei Bedarf ausgewertet werden.

Canceln eines Threads

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

volatile int x_global = 0;

void* run(void* arg) {
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, NULL);
    while (1) {
        int x_lokal = rand();
        x_global = x_lokal;
        if (x_lokal != x_global)
            printf("UPS: lokal=%d; global=%d\n", x_lokal, x_global);
        pthread_testcancel();
    }
}
```

Canceln eines Threads (Forts.)

```
int main() {
    srand(time(NULL));
    pthread_t t1, t2;
    pthread_create(&t1, NULL, run, NULL);
    pthread_create(&t2, NULL, run, NULL);

    sleep(30);

    pthread_cancel(t1);
    pthread_cancel(t2);
    void* res;
    pthread_join(t1, &res);
    if (res==PTHREAD_CANCELED) printf("abgebrochen\n");
    pthread_join(t2, NULL);
}
```

In diesem Beispiel wird das Cancel-Signal solange verzögert, bis in der Thread-Routine ein Cancelpunkt erreicht ist.