Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Wintersemester 2012/2013

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.

Kritische Bereiche

Threads teilen sich neben Prozess-ID, PPID, Filedeskriptoren, Terminal auch die globalen Variablen.
Lokale Variabeln der Thread-Routine sind hingegen individuell je Thread.

Im vorigen Beispiel teilen sich daher beide Threads `x_global`, hingegen besitzen beide Threads eine eigene Variable `x_lokal`.

Hierbei besitzt die Thread-Routine einen kritischen Bereich: Solange die `if`-Anweisung direkt nach der Zuweisung `x_global = x_lokal` erfolgt, darf diese niemals ausgeführt werden. Findet dazwischen aber eine Threadumschaltung statt, ist dies nicht mehr gewährleistet.
Daher erfolgen bei Ausführung gelegentlich Meldungen über ungleiche Werte.

Mutex

Zum Schützen eines kritischen Bereiches kann ein Mutex (für *mutual exclusion*) verwendet werden:

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Hiermit wird zu Beginn des kritischen Bereiches auf den Mutex-Lock gewartet und dieser am Ende wieder freigegeben. Bei der `trylock`-Variante wird auch direkt zurückgekehrt, wenn der Mutex gerade belegt ist (dann Rückgabewert ungleich Null, `errno` auf `EBUSY` gesetzt).

Beispiel mit Mutex

Das vorige Beispiel wird nun folgendermaßen geändert, um den kritischen Bereich zu schützen:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

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


Verklemmungen

Solange es nur einen Lock bzw. nur ein Objekt gibt, über das kritische Bereiche synchronisiert werden, tauchen keine Probleme auf.

Gibt es aber z. B. mehrere Locks, so kann es in der Praxis zu *Verklemmungen* bzw. *deadlocks* kommen. Hier blockiert das Programm (oder ein Teil des Programms), wenn es zu einer Situation kommt, wo Threads einen Lock besitzen und darauf warten, dass ein anderer frei wird, dieser jedoch von einem Thread gehalten wird, der wieder darauf wartet, dass ein anderer frei wird, . . .

Das Philosophenproblem

Eine solche Verklemmung wird musterhaft durch das *Philosphenproblem* beschrieben:

- Mehrere Philosophen sitzen an einem runden Tisch und wollen Spaghetti essen.
- Zwischen zwei Philosophen liegt jeweils eine Gabel, die beide sich teilen.
- Zum Essen laufen folgende Schritte ab:
  - Der Philosoph nimmt die linke Gabel in die Hand (ggfls. wartet er, bis sie frei ist),
  - dann nimmt er die rechte Gabel in die Hand (sobald sie frei ist),
  - er isst, bis er satt ist,
  - beide Gabeln werden wieder abgelegt.
- Nach dem Essen wird jeweils eine Denkpause eingelegt, bis der Philosoph wieder hungrig wird.

Schaubild

Bild: Benjamin D. Esham, Wikipedia

Implementierung

Im folgenden wird das Philosophenproblem mit zwei Philosophen implementiert, wobei die Gabeln jeweils ein Mutex sind:

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

#define MAX 2

typedef struct {
    int nr;
    pthread_mutex_t* links;
    pthread_mutex_t* rechts;
} gabel_t;
```

Implementierung (Forts.)

```
void* run(void* arg) {
    gabel_t* gabeln = (gabel_t*) arg;
    printf("Philosoph Nr. %d angelegt\n", gabeln->nr);
    while (1) {
        pthread_mutex_lock(gabeln->links);
        printf("Philosoph Nr. %d links aufgenommen\n", gabeln->nr);
        usleep(rand()%1000);
        pthread_mutex_lock(gabeln->rechts);
        printf("Philosoph Nr. %d rechts aufgenommen\n", gabeln->nr);
        usleep(10000+rand()%1000);
        pthread_mutex_unlock(gabeln->links);
        pthread_mutex_unlock(gabeln->rechts);
        printf("Philosoph Nr. %d fertig\n", gabeln->nr);
        usleep(10000+rand()%1000);
    }
}
```


Implementierung (Forts.)

```
int main() {
    srand(time(NULL));
    int i;
    pthread_t philosophen[MAX];
    pthread_mutex_t mutexe[MAX];
    gabel_t gabeln[MAX];

    for (i=0; i<MAX; i++) {
        pthread_mutex_init(&mutexe[i], NULL);
    }

    for (i=0; i<MAX; i++) {
        gabeln[i].nr = i+1;
        gabeln[i].links  = &mutexe[i];
        gabeln[i].rechts = &mutexe[(i+1)%MAX];
        pthread_create(&philosophen[i], NULL, run, (void*) &gabeln[i]);
        usleep(50000);
    }
}
```


Lösungsmöglichkeit

Die Ursache für die Verklemmung in diesem Fall liegt in der Reihenfolge des Aufnehmens der Gabeln. Um diese aufzulösen, kann man eine *Goldene Gabel* einführen, die grundsätzlich zuerst aufgenommen werden muss, auch wenn sie rechts liegt.

Übrigens reicht auch bei mehr als zwei Philosophen eine einzige goldene Gabel aus.

Dies bedeutet:
Letztlich ist ein einziger Philosoph (der links von der goldenen Gabel) dafür verantwortlich, ob *alle* Philosophen verhungern müssen oder nicht!

Lösung

```
pthread_mutex_t* golden = NULL;

void* run(void* arg) {
    gabel_t* gabeln = (gabel_t*) arg;
    pthread_mutex_t *first, *second;
    if (gabeln->rechts == golden) {
        first  = gabeln->rechts;
        second = gabeln->links;
    } else {
        first  = gabeln->links;
        second = gabeln->rechts;
    }
    while (1) {
        pthread_mutex_lock(first);
        usleep(rand()%1000);
        pthread_mutex_lock(second);
        ...
    }
}
```

In `main` wird dann eine der Gabeln markiert, z. B. durch

```
golden = &mutexe[0];
```