Linux – Prinzipien und Programmierung

Dr. Klaus Höppner

Hochschule Darmstadt – Sommersemester 2014

Kritische Bereiche bei Threads

Deadlocks

Conditions/Semaphore

Beispiel aus der letzten Vorlesung

```
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();
    }
}

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);
    pthread_join(t1, NULL); pthread_join(t2, NULL);
}
```

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];
```

Conditions

Wenn mehrere Threads synchronisiert ablaufen sollen, kann dies auch über *Conditions* geschehen, bei denen Threads blockiert werden, bis sie benachrichtigt werden, dass sie weiter machen sollen.

```
#include <pthread.h>

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex, const struct timespec *abstime);
```

Hierbei muss der Thread den angegebenen Mutex aktuell besitzen. Die erste Variante wartet beliebig lange auf die Benachrichtigung, die zweite maximal die angegebene Dauer. Im Fehlerfall ist der Rückgabewert ungleich 0 (und errno gleich `ETIMEDOUT` beim Timeout der zweiten Version).
Achtung: Der Thread gibt während des Wartens den Mutex wieder frei!

Benachrichtigung der wartenden Threads

Über die Condition können einer oder alle wartenden Threads benachrichtigt werden:

```
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Hierbei werden einer oder alle Threads aufgeweckt, wobei diese(r) dann wieder *auf den beim Wait-Befehl angegebenen Mutex wartet*.
Selbst im Fall eines Broadcasts über die Condition werden also i. A. nicht alle wartenden Threads gleichzeitig weiterarbeiten, sondern synchronsiert über den Mutex.

Erzeuger–Verbraucher über Ringpuffer

Im Folgenden sollen mehrere Erzeuger über einen Ringpuffer mehrere Verbraucher mit Daten (in diesem Fall zufällige Floats) versorgen.
Dieser Ringpuffer besteht aus einem Array und soll nach dem Prinzip FIFO arbeiten. Dafür merkt er sich jeweils die Position des ältesten Werts im Puffer und füllt diesen ringförmig, belegt also am Anfang des Array frei gewordene Plätze neu, nachdem beim Füllen das Array-Ende erreicht wurde.

Beim Einspeisen und Abholen von Werten aus dem Ringpuffer durch die Erzeuger bzw. Verbraucher werden folgende Fälle durch Conditions berücksichtigt:

- Der Ringpuffer ist leer, es stehen also keine Daten für Verbraucher zur Verfügung;
- der Ringpuffer ist voll, die Erzeuger können also keine Daten abliefern.

Teil 1: Definition des Ringpuffers

Datei ringbuffer.h:

```
#include <stddef.h>

typedef struct {
    float* buf;
    size_t capacity;
    size_t start;
    size_t count;
} rb_t;

int rb_init(rb_t *rb, size_t capacity);
int rb_destroy(rb_t *rb);
int rb_push(rb_t *rb, float value);
int rb_pop(rb_t *rb, float *value);
int rb_empty(rb_t *rb);
int rb_full(rb_t *rb);
```

Teil 2: Implementierung des Ringpuffers

Datei ringbuffer.c:

```
int rb_init(rb_t *rb, size_t capacity) {
    float *buf = calloc(capacity, sizeof(float));
    if (!buf) return(-1);
    rb->buf = buf;
    rb->capacity = capacity;
    rb->start = 0;
    rb->count = 0;
    return(0);
}

int rb_destroy(rb_t *rb) {
    if (rb->buf==NULL) return(-1);
    free(rb->buf);
    rb->buf=NULL;
    rb->capacity = 0;
    rb->start = 0;
    rb->count = 0;
    return(0);
}
```

Implementierung des Ringpuffers (Forts.)

```
int rb_push(rb_t *rb, float value) {
    if (rb->count>=rb->capacity) return(-1);
    size_t index = (rb->start + rb->count++) % rb->capacity;
    rb->buf[index] = value;
    return(0);
}

int rb_pop(rb_t *rb, float *value) {
    if (rb->count==0) return(-1);
    *value = rb->buf[rb->start++];
    rb->start %= rb->capacity;
    rb->count--;
    return(0);
}

int rb_empty(rb_t *rb) {
    return rb->count==0;
}

int rb_full(rb_t *rb) {
    return rb->count>=rb->capacity;
}
```

Logik der Conditions

Für die Erzeuger und Verbraucher werden nun folgender Mutex und folgende Conditions benutzt:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t data_available = PTHREAD_COND_INITIALIZER;
pthread_cond_t data_wanted = PTHREAD_COND_INITIALIZER;
```

Der Mutex wird von Erzeugern und Verbrauchern gemeinsam verwendet, wobei die Erzeuger im Falle eines vollen Puffers darauf warten, dass sie über die Condition `data_wanted` von einem Verbraucher benachrichtigt werden, dass er Daten abgeholt hat und nun wieder Platz frei ist. Im Fall des leeren Puffers warten die Verbraucher auf die Condition `data_available`, über die ein Erzeuger signalisiert, dass er Daten abgeliefert hat.

Teil 4: Erzeuger

```
void* produce(void* arg) {
    rb_t *buf = (rb_t*) arg;
    printf("Thread %lu ready to create data\n", pthread_self());
    while (1) {
        pthread_mutex_lock(&lock);
        while (rb_full(buf)) pthread_cond_wait(&data_wanted, &lock);
        float data = .0001F*rand();
        printf("Producer %lu created %f\n", pthread_self(), data);
        rb_push(buf, data);
        pthread_cond_broadcast(&data_available);
        pthread_mutex_unlock(&lock);
        usleep(1000);
    }
}
```

Teil 5: Verbraucher

```
void* consume(void* arg) {
    rb_t *buf = (rb_t*) arg;
    printf("Thread %lu ready to consume data\n", pthread_self());
    while (1) {
        pthread_mutex_lock(&lock);
        while (rb_empty(buf)) pthread_cond_wait(&data_available, &lock);
        float data;
        rb_pop(buf, &data);
        printf("Consumer %lu got %f\n", pthread_self(), data);
        pthread_cond_signal(&data_wanted);
        pthread_mutex_unlock(&lock);
        usleep(2000);
    }
}
```

Hauptprogramm

```
int main() {
    srand(time(NULL));
    rb_t buf;
    rb_init(&buf, 10);
    pthread_t e1, e2, v1, v2, v3;
    pthread_create(&v1, NULL, consume, &buf);
    pthread_create(&v2, NULL, consume, &buf);
    pthread_create(&v3, NULL, consume, &buf);
    pthread_create(&e1, NULL, produce, &buf);
    pthread_create(&e2, NULL, produce, &buf);

    sleep(10);
    pthread_cancel(v1);
    ...
    pthread_cancel(e2);
    pthread_join(v1, NULL);
    ...
    pthread_join(e2, NULL);
    rb_destroy(&buf);
    return(0);
}
```

Semaphore

Über einen Mutex konnte mit `lock` und `unlock` ein kritischer Bereich so geschützt werden, dass sich immer nur *ein* Thread in diesem Bereich befinden kann.

Über eine *Semaphore* kann ein Bereich für eine bestimmte Anzahl von Threads freigegeben werden. Technisch ist eine Semaphore eine (atomare) Ganzzahlvariable, die mit einem Wert größer Null initialisiert wird.
Der geschützte Bereich wird nur betreten, wenn die Semaphore größer Null ist (ansonsten wird gewartet) und diese dann heruntergezählt. Beim Verlassen des Bereichs wird die Semaphore wieder um eins erhöht.

Eine Semaphore mit der Maximalzahl 1 (*binary semaphore*) entspricht dabei einem Mutex.

POSIX-Semaphore

```
#include <semaphore.h>

int    sem_init(sem_t *s, int pshared, unsigned max);
int    sem_destroy(sem_t *s);

int    sem_wait(sem_t *s);
int    sem_timedwait(sem_t *s, const struct timespec * abstime);
int    sem_trywait(sem_t *s);
int    sem_post(sem_t *s);
```

Bei der Initialisierung wird die Maximalzahl von Threads festgelegt.
Mit den Wait-Funktionen wird beim Betreten des Bereichs die Semaphore dekrementiert, beim Post wieder inkrementiert.
Die drei Wait-Funktionen unterscheiden sich darin, ob beliebig lange, eine endliche Zeit oder gar nicht gewartet wird, bis ein Platz frei ist.

Beispiel

Hier haben in der Semaphore maximal 3 von 10 Threads Platz:

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

void* run(void *arg) {
    sem_t *s = (sem_t*) arg;
    while (1) {
        sem_wait(s);
        printf("Thread %lu in Semaphore\n", pthread_self());
        sleep(2);
        printf("Thread %lu verlaesst Semaphore\n", pthread_self());
        sem_post(s);
        sleep(1);
    }
}
```

Beispiel (Forts.)

```
int main() {
    sem_t sem;
    sem_init(&sem, 0, 3);

    pthread_t threads[10];
    int i;
    for (i=0; i<10; i++) pthread_create(&threads[i], NULL, run, &sem);
    sleep(10);
    for (i=0; i<10; i++) {
        pthread_cancel(threads[i]);
        pthread_join(threads[i], NULL);
    }
    sem_destroy(&sem);

    return(0);
}
```